

Scaling Byzantine Fault Tolerance

Thèse N° 9605

Présentée le 6 septembre 2019
à la Faculté informatique et communications
Laboratoire de calcul distribué
Programme doctoral en informatique et communications

pour l'obtention du grade de Docteur ès Sciences

par

Matej PAVLOVIČ

Acceptée sur proposition du jury

Prof. A. Ailamaki, présidente du jury
Prof. R. Guerraoui, directeur de thèse
Prof. I. Keidar, rapporteuse
Dr M. Vukolić, rapporteur
Prof. V. Kunčak, rapporteur

2019

Acknowledgements

I would like to thank everybody who helped me in my efforts towards this thesis. My parents and family who made it possible for me to study and supported me all the way through, all my friends who helped me stay sane when I was not currently working, and my teachers and sports trainers who taught me to work hard towards my goals.

In research specifically, I want to thank, in the first place, my supervisor, Prof. Rachid Guer-raoui, for accepting me as his student, guiding me throughout all my doctoral studies, for trusting and believing in me and for opening many doors for me. The content of this thesis is also heavily based on common efforts together with my colleagues and collaborators, who contributed not only by sharing ideas and insights, but also by hours spent proofreading and correcting texts, providing feedback and in general by creating a great working environment. I also wish to thank the members of my jury, Idit Keidar, Marko Vukolic, and Viktor Kuncak, for their feedback and improvement suggestions.

My doctoral studies were financed by the Swiss National Science Foundation (project 200021_147067).

The possibility to even start working towards a PhD was given to me by the Mondi Austira Studen Scholarship by fully financing my bachelor's and master's studies.

This thesis could not have been written without all those just mentioned.

Lausanne, 25 June 2019

Preface

This thesis contains selected results of research performed during doctoral studies between September 2013 and March 2019, supervised by Prof. Rachid Guerraoui, at the Distributed Computing Laboratory of the School of Computer and Communication Sciences at EPFL in Lausanne, Switzerland. The results presented by this thesis appear in the following works¹:

- Rachid Guerraoui, Petr Kuznetsov, Matteo Monti, Matej Pavlovic, Dragos-Adrian Seredinschi. “The Consensus Number of a Cryptocurrency”. *PODC 2019*.
- Rachid Guerraoui, Petr Kuznetsov, Matteo Monti, Matej Pavlovic, Dragos-Adrian Seredinschi. “Scalable Secure Broadcast”. *Under submission*.
- Rachid Guerraoui, Anne-Marie Kermarrec, Matej Pavlovic, Dragos-Adrian Seredinschi. “Atum: Scalable Group Communication Using Volatile Groups”. *Middleware 2016*.

The following additional publications contain work that is not presented in this thesis, but are also a result of research performed during the same doctoral studies.

- Oana Balmau, Rachid Guerraoui, Anne-Marie Kermarrec, Alexandre Maurer, Matej Pavlovič, Willy Zwaenepoel.¹ “The Fake News Vaccine”. *NETYS 2019*.
- Rachid Guerraoui, Petr Kuznetsov, Matteo Monti, Matej Pavlovič, Dragos-Adrian Seredinschi.¹ “AT2: Asynchronous Trustworthy Transfers”. <https://arxiv.org/abs/1812.10844>, 2018.
- Rachid Guerraoui, Matej Pavlovič, Dragos-Adrian Seredinschi.¹ “Blockchain Protocols: The Adversary is in the Details”. *Symposium on Foundations and Applications of Blockchain 2018*.
- Matej Pavlovič, Alex Kogan, Virendra J. Marathe, Tim Harris. “Brief Announcement: Persistent Multi-Word Compare-and-Swap”. *PODC 2018*.
- Yihe Huang, Matej Pavlovič, Virendra Marathe, Margo Seltzer, Tim Harris, Steve Byan. “Closing the Performance Gap Between Volatile and Persistent Key-Value Stores Using Cross-Referencing Logs”. *USENIX ATC 2018*.
- Rachid Guerraoui, Matej Pavlovič, Dragos-Adrian Seredinschi.¹ “Trade-offs in Replicated Systems”. *IEEE Data Engineering Bulletin 39(1) 2016*.

Preface

- Rachid Guerraoui, Matej Pavlovič, Dragos-Adrian Seredinschi.¹ “Incremental Consistency Guarantees for Replicated Objects”. *OSDI 2016*.

¹Author names appear in alphabetical order.

Abstract

Online services are becoming more and more ubiquitous and keep growing in scale. At the same time, they are required to be highly available, secure, energy-efficient, and to achieve high performance. To ensure these (and many other) properties, replication and distribution of these services becomes inevitable. Indeed, today's online services often involve thousands of processes running on different machines interconnected by a communication network. These processes may experience various kinds of failures, from simply crashing to being compromised by a malicious (Byzantine) adversary. The classic method for dealing with Byzantine faults is state machine replication (SMR). However, SMR fundamentally relies on a solution of the consensus problem, which often proves to be a scalability bottleneck.

This dissertation addresses the scalability of Byzantine fault-tolerant systems. We argue that, for a certain class of applications, consensus either does not need to be solved at all, or only needs to be solved among a limited number of processes. By circumventing the consensus problem where solving it is not necessary, we improve the scalability of these applications.

We start by focusing on the particular problem of distributed asset transfer, where digital assets are being transferred between user accounts—a problem underlying many cryptocurrency systems, most of which address it using Byzantine fault-tolerant SMR (and thus consensus). We show that consensus is not required for asset transfer by defining it as a sequential object type in the shared memory model and proving that it has consensus number 1 in Herlihy's hierarchy. We further generalize the asset transfer object type, allowing an account to be shared by up to k owners. We prove that the consensus number of such an object type is k .

We also discuss the asset transfer problem in the message passing model. We devise a consensusless asset transfer algorithm that relies on a secure broadcast primitive that, unlike consensus, has fully asynchronous deterministic implementations.

Furthermore, since deterministic implementations of secure broadcast have limited scalability, we propose probabilistic secure broadcast, a variant of secure broadcast where some properties are allowed to be violated with a bounded probability. We design a highly scalable randomized algorithm that implements probabilistic secure broadcast with an arbitrarily low bound on the failure probability.

Finally, we present Atum, a system for scalable group communication in a Byzantine environment that supports high churn. Atum achieves scalability by partitioning the system into groups of logarithmic size, only executing a consensus protocol inside each group.

Preface

Keywords: distributed systems, Byzantine fault tolerance, scalability, replication, consensus, asset transfer, secure broadcast, randomized algorithms, group communication, peer-to-peer systems

Zusammenfassung

Online Services sind omnipräsent und werden im immer größeren Ausmaß eingesetzt. Gleichzeitig wird von diesen Services hohe Verfügbarkeit, Sicherheit, Energieeffizienz und Leistung gefordert. Um diese (und viele andere) Eigenschaften zu erreichen, müssen die Services notwendigerweise repliziert und verteilt sein. Heutige online Services umfassen tatsächlich tausende Prozesse, die verteilt auf vielen, durch ein Netz kommunizierenden Maschinen ausgeführt werden. Dabei können die verschiedensten Fehler auftreten, von einfachen Maschinenabstürzen bis zu sogenannten Byzantinischen Fehlern. Eine Standardmethode beim Bekämpfen Byzantinischer Fehler ist “state machine replication” (SMR). Diese Methode basiert aber grundsätzlich auf dem Lösen des Konsensusproblems, was oft die Skalierbarkeit des Systems beeinträchtigt.

Diese Dissertation befasst sich mit der Skalierbarkeit von Byzantinisch fehlertoleranten Systemen. Wir zeigen, dass für eine Bestimmte Klasse von Anwendungen das Lösen des Konsensusproblems entweder nur in einer beschränkten Gruppe von Prozessen, oder sogar überhaupt nicht notwendig ist. Das Vermeiden vom Konsensusproblem wo möglich trägt zur Skalierbarkeit der entsprechenden Systeme bei.

Wir konzentrieren uns zuerst auf eine spezifische Anwendung—die Überweisung von Gütern (z.B. Geld) zwischen Benutzerkonten. Dies ist eine Anwendung von vielen Kryptowährungen und wird meistens mit Hilfe von SMR (und damit Konsensus) implementiert. Wir zeigen, dass Konsensus hier unnötig ist, indem wir das Überweisungsproblem als ein Objekttyp im Shared Memory definieren und beweisen, dass in Herlihy's Hierarchie seine Konsensusnummer 1 ist. Wir verallgemeinern diesen Objekttyp, sodass ein Konto bis zu k Eigentümer haben kann, und wir beweisen dass so ein Objekttyp Konsensusnummer k hat.

Wir stellen auch einen auf Nachrichtenaustausch basierenden Überweisungsalgorithmus vor, der auf Broadcast beruht und, im Gegensatz zu Konsensus, asynchron und deterministisch implementiert werden kann.

Da eine deterministische Implementierung der dazu benötigten Broadcastvariante nur eingeschränkt skalierbar ist, stellen wir eine probabilistische Alternative vor, die zwar eine (beliebig kleine) Wahrscheinlichkeit von Fehler zulässt, dafür aber höchst skalierbar ist.

Schließlich präsentieren wir Atum ein, dynamisches, Byzantinisch fehlertolerantes “group communication system”. Wir erreichen Skalierbarkeit, indem wir das System in (logarithmisch) kleine Prozessgruppen unterteilen und Konsensus nur innerhalb dieser Gruppen benötigen.

Preface

Schlüsselwörter: verteilte Systeme, Byzantinische Fehlertoleranz, Skalierbarkeit, Replikation, Konsensus, Güterüberweisung, secure broadcast, Randomisierte Algorithmen, group communication, peer-to-peer Systeme

Contents

Acknowledgements	iii
Preface	v
Abstract (English/Deutsch)	vii
List of figures	xiii
List of algorithms	xv
1 Introduction	1
2 Asset Transfer in Shared Memory	5
2.1 Introduction	6
2.2 Shared Memory Model and Asset-Transfer Object Type	8
2.2.1 Shared memory	8
2.2.2 The asset-transfer object type	9
2.3 Asset-Transfer Has Consensus Number 1	10
2.4 k -Shared Asset-Transfer Has Consensus Number k	13
3 Asset Transfer in Message Passing	21
3.1 Introduction	21
3.2 Byzantine-Tolerant Asset Transfer	22
3.3 Asset Transfer Implementation in Byzantine Message Passing	23
3.4 k -shared Asset Transfer in Message Passing	29
4 Probabilistic Secure Broadcast	33
4.1 Introduction	33
4.1.1 Samples	34
4.1.2 Scalable Secure Broadcast	35
4.2 Model and Assumptions	37
4.3 Probabilistic broadcast	38
4.3.1 Definition	39
4.3.2 Algorithm	39
4.3.3 Correctness	41
	xi

Contents

4.4	Probabilistic consistent broadcast	42
4.4.1	Definition	43
4.4.2	Algorithm	43
4.4.3	Correctness	46
4.5	Probabilistic secure broadcast	47
4.5.1	Definition	48
4.5.2	Algorithm	48
4.5.3	Correctness	52
5	Atum: Scalable Group Communication Using Volatile Groups	57
5.1	Introduction	57
5.2	Assumptions and Guarantees	60
5.3	Design	61
5.3.1	Group layer	63
5.3.2	Overlay layer	65
5.3.3	API operations	67
5.4	Applications	69
5.4.1	ASub	69
5.4.2	AShare	70
5.4.3	AStream	72
5.5	Deploying Atum	73
5.5.1	Practical considerations	73
5.5.2	Atum implementations	75
5.6	Evaluation	75
5.6.1	Base evaluation of Atum	75
5.6.2	Evaluating AShare	79
5.6.3	Evaluating AStream	81
5.7	Experiences and Lessons Learned	81
5.8	Conclusions	83
6	Related Work	85
6.1	Asset Transfer	85
6.2	Group Communication	86
6.2.1	Broadcast abstractions	87
6.2.2	Gossip	87
6.2.3	Robust overlay networks	88
6.2.4	Membership sampling	88
6.2.5	Storage	89
7	Conclusions	91
	Bibliography	103

List of Figures

3.1 High-level structure of our message-passing implementation of asset transfer. The upper layer (whose algorithm is depicted in Algorithm 5) uses a secure broadcast abstraction as a black box.	25
4.1 Layered view of our broadcast abstractions. Starting from authenticated point-to-point links, we implement secure broadcast using 3 layers of broadcast abstractions. Intuitively, each layer is mainly responsible for guaranteeing one of secure broadcast's properties	36
5.1 Atum's layered architecture.	62
5.2 An instance of Atum: Vgroups interconnected by an H-graph overlay with two cycles.	62
5.3 Two vgroups communicate (e.g., gossiping) through a group message, which consists of multiple inter-node messages.	64
5.4 Guideline with optimal rw_1 and hc system parameters.	66
5.5 AShare: A feedback loop triggers the randomized replication algorithm repeatedly. c is the number of replicas for a file.	71
5.6 Growth speed for systems with up to 1400 nodes.	76
5.7 Maximal tolerated churn rates in systems of size 50, 100, 200, 400 and 800 nodes.	77
5.8 Group communication latency: Comparison between gossip, Atum, and SMR. We tag with * the systems with 50 faults.	78
5.9 AShare: Read performance (latency per MB). We normalize the result to file size.	79
5.10 AShare: Impact of Byzantine nodes on read latency. Experiment with 50 nodes (7 Byzantine) and 500 files.	80
5.11 AShare: Impact of Byzantine nodes on read latency. Experiment with 100 nodes (7 Byzantine) and 1000 files.	80
5.12 AStream: Latency for 1MB/s data stream.	81
5.13 As a system grows faster, the quality of random vgroup composition suffers due to suppressed exchanges.	82

List of Algorithms

1	Wait-free implementation of asset-transfer: code for process p	11
2	Wait-free implementation of consensus among k processes using a k -shared asset-transfer object and read-write registers. Code for process $p \in \{1, \dots, k\}$. . .	13
3	Wait-free implementation of a k -shared asset-transfer object using k -consensus objects. Code for process p	15
4	Auxiliary functions used by algorithm in Algorithm 3	16
5	Consensusless transfer system based on secure broadcast. Code for process p	26
6	Erdős-Rényi Gossip	40
7	Procedure <i>sample</i>	43
8	Echo Broadcast	44
9	Ready Broadcast	49

1 Introduction

More and more tasks are being performed and services are being provided by computer systems. These tasks are becoming more and more complex and the underlying services supporting these tasks are being used by an ever-increasing number of entities, be they human users or other computer systems. Such services are often subject to stringent requirements in performance, availability, security, energy-efficiency, and others.

For many services regularly used today, it has become unthinkable to be deployed on single machines. Instead, mostly for performance and availability reasons, many services must be deployed as distributed systems. These systems often consist of thousands of communicating processes, possibly distributed all across the planet. If several processes in such a distributed system deal with the same service state, they must coordinate to keep this state consistent, even if none of the processes failed. Failures, however, become inevitable in such scenarios and must be considered the normal case rather than a rare exception. To prevent the service from becoming unavailable in the event of a failure, it must be replicated, making consistency even harder to achieve. Fault tolerance of distributed systems is full of challenges and is therefore still an active area of research.

A particular concern is Byzantine fault tolerance. A Byzantine fault-tolerant distributed system guarantees correct operation despite arbitrary behavior of some of its processes.¹ Unlike a crash fault, where a process is assumed to either work correctly or stop operating altogether, there are, strictly speaking, no assumptions whatsoever on the behavior of a Byzantine-faulty process (even though, in practice, the resources—e.g. computational ones—of Byzantine-faulty processes are usually assumed not to be unlimited).

A process might deviate from a correct behavior due to various reasons. These reasons include software bugs, unlikely but possible spontaneous random bit flips in memory chips or erroneous inputs. Moreover, a process might also misbehave if it becomes compromised by a malicious adversary whose goal is to subvert the functioning of the system.

¹Describing arbitrary behavior by the term Byzantine has been introduced by Lamport et al. in their seminal paper “The Byzantine Generals Problem” [LSP82].

Chapter 1. Introduction

Especially with the rise of Bitcoin [Nak08] and other cryptocurrencies, distributed peer-to-peer systems with limited or no trust between the participating processes gain on relevance. Such systems need to both accommodate large numbers of participating processes and be able to tolerate Byzantine failures of some of them. Thus, unsurprisingly, scalable Byzantine fault tolerance is regaining attention from the research community.

We focus on the technique of replication, which occurs, in one way or another, in almost any large-scale distributed system. In particular, we study how replication can be done in a Byzantine fault-tolerant way that scales to large numbers of processes.

The canonical way of replicating an on-line service is through state machine replication [Lam78, Sch90]. The service is modeled as a deterministic state machine consisting of a state and a set of operations that may alter that state. Multiple instances of this (logical) state machine are deployed on different replicas (physical machines) and a distributed protocol makes sure that those machines execute the same sequence of operations on the same initial state. To this end, it is necessary for all replicas to *agree* on the order in which to execute operations. All operations being deterministic, they produce the same outputs at every replica (generalizations of this approach exist for replicating even non-deterministic state machines [CSV17]).

State machine replication is an approach that is *universal*, in the sense that it can be used to replicate *any* service that can be modeled as a state machine (which is generally the case). Many distributed protocols exist for implementing state machine replication in various scenarios, putting various assumptions on the possible behavior of processes and on the network through which the processes communicate. There are standard solutions to state machine replication both in the context of crash-only faults, as well as in the context of Byzantine faults.

However, universality comes at a cost. This is partly due to the fact that to obtain a total order of operations, such universal solutions fundamentally require solving the central problem of distributed computing—the consensus problem. This makes state machine replication notoriously difficult to implement correctly, even proven impossible under certain conditions [FLP85, AGK⁺15]. Making protocols Byzantine fault-tolerant only adds more complexity to an already challenging task. Working solutions often only scale to a few replicas, their protocol overhead quickly becoming prohibitive as the number of replicas increases. Efforts to address this problem, such as Honey Badger BFT [MXC⁺16], are rather recent.

We examine alternative replication schemes that either avoid state machine replication altogether, or confine it to a part of the system that is only as small as necessary to perform a given task. Many applications of practical relevance can indeed be replicated without the need for universal agreement among the replicas.

Our main focus is on the application of distributed asset transfer, also known as a cryptocurrency, that has been receiving much attention recently both from academia and from industry. We show that distributed secure asset transfer (and similar problems) can be solved without the recourse to agreement and state machine replication. We achieve this by exploiting the fact

that only the owner of an asset can transfer it to a new owner. Instead of agreeing on the order of operations to execute, it is sufficient to broadcast operations in a secure and consistent way among the replicas.

While broadcasting operations is simpler and cheaper than agreeing on a unique order of those operations, efficiently scaling it to a large system in a Byzantine fault-prone environment is a non-trivial task. Using randomization, however, we are able to provide a highly scalable broadcast algorithm for the price of a fixed failure probability that can be made arbitrarily small.

Furthermore, we show how to replicate services that are more powerful than asset transfer, in the sense that they do require agreement at least among certain processes in the system. We only execute an agreement protocol where absolutely necessary, avoiding the scalability bottleneck of agreement across the whole system. In order to ensure that a group of processes can reach agreement, we use randomization to make sure that most of the processes in such a group are correct with high probability.

The contributions in this thesis are the following.

Asset Transfer and Consensus

- We formally define the asset transfer problem as a shared object in the shared memory model.
- We prove that asset transfer has consensus number 1 in Herlihy's consensus hierarchy [Her91] by implementing asset transfer in shared memory using an atomic snapshot object [AAD⁺93]. This means that solving consensus is not necessary in order to implement asset transfer.
- We generalize the asset transfer problem by allowing an account to have multiple owners that can all atomically perform transfers from their shared account. We call this generalized variant the k -shared asset transfer if each account has at most k owners.
- We prove that k -shared asset transfer has consensus number k in Herlihy's consensus hierarchy, meaning that k processes can solve the consensus problem using k -shared account objects. Our proof is based on reducing k -shared asset transfer to the k -consensus problem (known for having consensus number k) and vice versa.
- We provide an algorithm that implements asset transfer in the message passing model using a secure broadcast primitive [BT85a].
- We provide a novel scalable algorithm for secure broadcast that is used for implementing asset transfer. Relying on randomization, our algorithm allows a certain probability of failure. However, this probability can be made arbitrary small. In a system of N

processes, our algorithm has $O(\log(n))$ per node communication complexity, allowing it to scale to large system sizes.

Scalable Group Communication Using Volatile Groups

- We introduce the notion of volatile groups—a partitioning of the system in small groups, only executing an agreement protocol within each group. By randomizing the composition of the groups even in the presence of high churn (processes leaving and joining), we make sure that, with high probability, a malicious adversary is unable to subvert any group by making too many faulty processes join the same group.
- Using these volatile groups, we design Atum, a group communication system for large-scale, dynamic and Byzantine fault-prone environments.
- We implement two variants of Atum, one for the synchronous and one for the eventually synchronous system model.
- We evaluate Atum using three applications built on top of it.

2 Asset Transfer in Shared Memory

In this chapter we closely examine *asset transfer* as a stand-alone problem. As a starting point, we consider asset transfer systems, often implemented by blockchain-based algorithms, such as Bitcoin. Such systems are often referred to as *cryptocurrencies*.

As stated in the original paper by Nakamoto [Nak08], at the heart of these systems lies the problem of preventing any participant from engaging in *double-spending*, i.e., spending the same asset more than once. This is usually solved by achieving *consensus* on the order of transfers among the participants. We show that consensus is not necessary to prevent double-spending by defining the asset transfer problem as a *concurrent object* in the shared memory model and determining its *consensus number*.

We first consider the problem as defined by Nakamoto, where only a single process—the account owner—can withdraw from each account. We prove that the consensus number of an asset transfer object is 1 by reducing asset transfer to an atomic snapshot object that is known to have a wait-free implementation in read-write shared memory.

We then consider a more general *k-shared* asset transfer object where up to k processes can atomically withdraw from the same account, and show that this object has consensus number k .

We prove the lower bound by providing a wait-free implementation of consensus in a shared memory system with k processes equipped with a k -shared asset transfer object. Each process first announces its proposed value in a dedicated (per-process) register, and then tries to perform a transfer from a shared account. We choose the initial account balance and the withdrawn amount such that (1) only one withdrawal is possible and (2) the remaining balance identifies the withdrawing process. Our upper bound proof, borrowing concepts from universal constructions, consists of reducing k -shared asset transfer to k -consensus.

2.1 Introduction

In 2008, Satoshi Nakamoto introduced the Bitcoin protocol, implementing an electronic decentralized asset transfer system, often called a cryptocurrency [Nak08]. Since then, many alternatives to Bitcoin came to prominence. These include major cryptocurrencies such as Ethereum [Woo15] or Ripple [RLGS14], as well as systems sparked from research or industry efforts such as Bitcoin-NG [EGSVR16], Algorand [GHM⁺17], ByzCoin [KJG⁺16], Stellar [Maz15], Hyperledger Fabric [ABB⁺18], Corda [Hea16], or Solida [AMN⁺16]. Each alternative brings novel approaches to implementing decentralized transfers, and sometimes offers a more general interface (known as smart contracts [Sza97]) than the original protocol proposed by Nakamoto. They improve over Bitcoin in various aspects, such as performance, energy-efficiency, or security.

A common theme in these protocols, whether they are for transfers [KKJG⁺18] or smart contracts [Woo15], is that they usually implement some form of a *blockchain*—a distributed ledger where all the transfers in the system are totally ordered. Achieving total order among multiple inputs (e.g., transfers) is fundamentally a hard task, equivalent to solving *consensus* [FLP85, HT93]. Consensus, one of the most important problems in distributed computing, is well-known to be difficult to solve. It has been proven that a deterministic solution in an asynchronous system does not exist as long as only a single participant can fail by crashing. [FLP85]. It is highly non-trivial to devise even partially synchronous consensus algorithms that work correctly [AGM⁺17, CV17, CWA⁺09, AGK⁺15] and existing solutions face tough dilemmas between security, energy-efficiency, and performance [AGMS18, BGP89, GPS18, Vuk15]. Not surprisingly, the consensus module is a major bottleneck in blockchain-based protocols [Hea16, SBV18, Vuk15]. Numerous solutions have emerged to alleviate this problem [GAG⁺18, Hea16]. Typical techniques seek to employ a form of sharding [KKJG⁺18], for instance, or to use a committee-based optimization [EGSVR16, GHM⁺17]. We circumvent this main bottleneck, yielding new solutions that bypass consensus altogether.

A closer look at Nakamoto’s original paper [Nak08] reveals that the main problem underlying a decentralized asset transfer system (i.e., a cryptocurrency) is preventing *double-spending*. Double-spending is the event of a malicious participant transferring an asset, and subsequently (or concurrently) transferring the same asset to a potentially different destination. Bitcoin and follow-up systems typically assume that total order—and thus consensus—is vital to preventing double-spending [GKL15]. There seems to be a common belief, indeed, that a consensus algorithm is essential for implementing decentralized asset transfers [BMC⁺15, GPS18, KJW⁺18, Nak08].

We show that this belief is false by casting the asset transfer problem as a *sequential object type* and determining that it has *consensus number 1* in Herlihy’s hierarchy [Her91]. (The consensus number of an object type is the maximal number of processes that can solve consensus using only read-write shared memory and arbitrarily many objects of this type.)

Intuitively, an asset transfer object consists of accounts whose balances can be read by all

processes, and where processes are allowed to transfer assets between accounts. Each account is associated with an *owner* process that is the only one allowed to issue transfers withdrawing from this account. Our result is based on the insight that this association of accounts to unique owners obviates the need for consensus. It is the owner that decides on the order of transfers from its own account, without the need to agree with any other process—thus the consensus number 1. Other processes only validate the owner’s decisions, ensuring that causal relations across accounts are respected.

In our asset transfer implementation, processes share the performed transfer operations using atomic snapshot memory [AAD⁺93]. Each process validates a transfer outgoing from an account by relating the withdrawn amount with the transfers found in the memory snapshot that are incoming to that account. At most one withdrawal can be active on a given account at a time, since we define processes to be sequential and each account has a single owner. It is thus safe to declare the validated operation as successful and add it in the snapshot memory.

Our result naturally generalizes to the setting in which multiple processes are allowed to withdraw from the same account. A *k-shared* asset transfer object allows up to k processes to execute outgoing transfers from the same account. We prove that such an object has consensus number k using k -consensus objects [JT92]. This means that k -shared asset transfer allows for implementing *state machine replication* (now often referred to as *smart contracts*) among the k involved processes.

We show that k -shared asset transfer has consensus number at most k by reducing it to k -consensus, known to have consensus number k . Our reduction borrows concepts from universal construction of sequential objects on top of consensus: the k owners of an account use a sequence of k -consensus objects to agree on the order of outgoing transfers and their results. We then show that k -shared asset transfer has consensus number at least k by devising an algorithm where up to k processes solve consensus by transferring distinct amounts from the same k -shared account in an asset transfer object.

In the k -shared case, our result implies that to execute some form of *smart contract* involving k users, consensus is only needed among these k processes and not among all processes in the system. In particular, should these k processes be faulty, the rest of the accounts will not be affected.

To summarize, we argue that treating *double-spending* as a distributed computing problem and measuring its hardness through the lenses of distributed computing metrics help understand it and devise better solutions to it.

The rest of this chapter is organized as follows. We first give the formal definition of the shared memory model and the asset transfer object type (Section 2.2). Then, we show that it has consensus number 1 (Section 2.3). Finally, we generalize our result by proving that a k -shared asset transfer object has consensus number k (Section 2.4).

2.2 Shared Memory Model and Asset-Transfer Object Type

We now present the shared memory model (Section 2.2.1) and precisely define the problem of asset-transfer as a sequential object type (Section 2.2.2).

2.2.1 Shared memory

Processes. We assume a set Π of N asynchronous processes that communicate by invoking atomic operations on shared memory objects. Processes are sequential—we assume that a process never invokes a new operation before obtaining a response from a previous one.

Object types. A sequential object type is defined as a tuple $T = (Q, q_0, O, R, \Delta)$, where Q is a set of states, $q_0 \in Q$ is an initial state, O is a set of operations, R is a set of responses and $\Delta \subseteq Q \times \Pi \times O \times Q \times R$ is a relation that associates a state, a process identifier and an operation to a set of possible new states and corresponding responses. Here we assume that Δ is total on the first three elements, i.e., for each state $q \in Q$, each process $p \in \Pi$, and each operation $o \in O$, some transition to a new state is defined, i.e., $\forall q \in Q, p \in \Pi, o \in O : \exists q' \in Q, r \in R : (q, p, o, q', r) \in \Delta$.

A *history* is a sequence of invocations and responses, each invocation or response associated with a process identifier. A *sequential history* is a history that starts with an invocation and in which every invocation is immediately followed with a response associated with the same process. A sequential history $(j_1, o_1), (j_1, r_1), (j_2, o_2), (j_2, r_2), \dots$, where $\forall i \geq 1, j_i \in \Pi, o_i \in O, r_i \in R$, is *legal* with respect to type $T = (Q, q_0, O, R, \Delta)$ if there exists a sequence q_1, q_2, \dots of states in Q such that $\forall i \geq 1, (q_{i-1}, j_i, o_i, q_i, r_i) \in \Delta$.

Implementations. An *implementation* of an object type T is a distributed algorithm that, for each process and invoked operation, prescribes the actions that the process needs to take to perform it. An *execution* of an implementation is a sequence of *events*: invocations and responses of operations or atomic accesses to shared abstractions. The sequence of events at every process must respect the algorithm assigned to it.

Failures. Processes are subject to *crash failures*.¹ A process may halt prematurely, in which case we say that the process is *crashed*. A process is called *faulty* if it crashes during the execution. A process is *correct* if it is not faulty. All algorithms we present in the shared memory model are *wait-free*—every correct process eventually returns from each operation it invokes, regardless of an arbitrary number of other processes crashing or concurrently invoking operations.

Linearizability. For each pattern of operation invocations, the execution produces a *history*, i.e., the sequence of distinct invocations and responses, labelled with process identifiers and

¹The assumption of crash-only failures is temporary—we only use it in the context of shared memory. Later on, in the message passing context, we consider Byzantine failures.

unique sequence numbers.

A projection of a history H to process p , denoted $H|p$ is the subsequence of elements of H labelled with p . An invocation o by a process p is *incomplete* in H if it is not followed by a response in $H|p$. A history is *complete* if it has no incomplete invocations. A *completion* of H is a history \bar{H} that is identical to H except that every incomplete invocation in H is either removed or *completed* by inserting a matching response somewhere after it.

An invocation o_1, r_1 *precedes* an invocation o_2 in H , denoted $o_1 <_H o_2$, if o_1 is complete and the corresponding response r_1 precedes o_2 in H . Note that $<_H$ stipulates a partial order on invocations in H . A *linearizable* implementation of T ensures that for every history H it produces, there exists a completion \bar{H} and a legal sequential history S such that (1) for all processes p , $\bar{H}|p = S|p$ and (2) $<_H \subseteq <_S$.

2.2.2 The asset-transfer object type

Let \mathcal{A} be a set of *accounts* and $\mu : \mathcal{A} \rightarrow 2^\Pi$ be an “owner” map that associates each account with a set of processes that are, intuitively, allowed to debit the account. We define the asset-transfer object type associated with \mathcal{A} and μ as a tuple (Q, q_0, O, R, Δ) , where:

- The set of states Q is the set of all possible maps $q : \mathcal{A} \rightarrow \mathbb{N}$. Intuitively, each state of the object assigns each account its *balance*.
- The initialization map $q_0 : \mathcal{A} \rightarrow \mathbb{N}$ assigns the initial balance to each account.
- Operations and responses of the type are defined as $O = \{\text{transfer}(a, b, x) : a, b \in \mathcal{A}, x \in \mathbb{N}\} \cup \{\text{read}(a) : a \in \mathcal{A}\}$ and $R = \{\text{true}, \text{false}\} \cup \mathbb{N}$.
- Δ is the set of valid state transitions. For a state $q \in Q$, a process $p \in \Pi$, an operation $o \in O$, a response $r \in R$ and a new state $q' \in Q$, the tuple $(q, p, o, q', r) \in \Delta$ if and only if one of the following conditions is satisfied:
 - $o = \text{transfer}(a, b, x) \wedge p \in \mu(a) \wedge q(a) \geq x \wedge ((q'(a) = q(a) - x \wedge q'(b) = q(b) + x \wedge a \neq b) \vee (q'(a) = q(a) \wedge a = b)) \wedge \forall c \in \mathcal{A} \setminus \{a, b\} : q'(c) = q(c)$ (all other accounts unchanged) $\wedge r = \text{true}$;
 - $o = \text{transfer}(a, b, x) \wedge (p \notin \mu(a) \vee q(a) < x) \wedge q' = q \wedge r = \text{false}$;
 - $o = \text{read}(a) \wedge q = q' \wedge r = q(a)$.

In other words, operation $\text{transfer}(a, b, x)$ invoked by process p *succeeds* if and only if p is the owner of the source account a and account a has enough balance, and if it does, x is transferred from a to the destination account b . A $\text{transfer}(a, b, x)$ operation is called *outgoing* for a and *incoming* for b ; respectively, the x units are called *outgoing* for a and *incoming* for b . A transfer is *successful* if its corresponding response is *true* and *failed* if its corresponding

response is *false*. Operation $read(a)$ simply returns the balance of a and leaves the account balances untouched.

As in Nakamoto’s original paper [Nak08], we assume that an asset-transfer object has at most one owner per account. Unless stated otherwise, we assume that $\forall a \in \mathcal{A} : |\mu(a)| \leq 1$. Later we lift this assumption and consider more general k -shared asset-transfer objects with arbitrary owner maps μ (Section 2.4). For the sake of simplicity, we also restrict ourselves to transfers with a single source account and a single destination account. However, the definition (and implementation) of the asset-transfer object type can trivially be extended to support transfers with multiple source accounts (all owned by the same process) and multiple destination accounts.

2.3 Asset-Transfer Has Consensus Number 1

In this section, we discuss the “universality” of the asset-transfer type, showing that it can be implemented in a *wait-free* manner using only read-write registers. Thus, the type has consensus number 1.

Consider an asset-transfer object associated with a set of accounts \mathcal{A} and an ownership map μ where $\forall a \in \mathcal{A}, |\mu(a)| \leq 1$. We now present our wait-free implementation of this object in the read-write shared-memory model.

Our implementation of asset-transfer is based on the atomic snapshot object that is known to have a wait-free implementation in read-write shared memory [AAD⁺93]. Such an object consists of entries, each of which is assigned to a single process that can write to it. Moreover, any process can atomically read the contents of all entries. More precisely, the atomic snapshot (AS) object is represented as a vector of N shared variables that can be accessed with two atomic operations: *update* and *snapshot*. An *update* operation modifies the value at a given position of the vector and a *snapshot* returns the state of the whole vector.

In our algorithm (described in Figure 1), the N processes share such an atomic snapshot object. We use the AS object to represent the state of the accounts in the asset-transfer object being implemented. Every process p is associated with a distinct location in the atomic snapshot object storing the set of all successful *transfer* operations executed by p so far. Since each account is owned by at most one process, all outgoing transfers for an account appear in a single location of the atomic snapshot (associated with the owner process). We implement the *read* and *transfer* operations as follows.

- To *read* the balance of an account a , the process simply takes a snapshot S and returns the initial balance plus the sum of incoming amounts minus the sum of all outgoing amounts found in all locations of the snapshot. We denote this number by $balance(a, S)$. As we argue below, the result is guaranteed to be non-negative, i.e., the operation is correct with respect to the type specification.

Algorithm 1 Wait-free implementation of asset-transfer: code for process p

Shared variables:

AS , atomic snapshot, initially $\{\perp\}^N$

Local variables:

$ops_p \subseteq \mathcal{A} \times \mathcal{A} \times \mathbb{N}$, initially \emptyset

Upon $transfer(a, b, x)$

```

1   $S = AS.snapshot()$ 
2  if  $p \notin \mu(a) \vee balance(a, S) < x$  then
3    return false
4   $ops_p = ops_p \cup \{(a, b, x)\}$ 
5   $AS.update(ops_p)$ 
6  return true

```

Upon $read(a)$

```

7   $S = AS.snapshot()$ 
8  return  $balance(a, S)$ 

```

- To perform $transfer(a, b, x)$, a process p , the owner of a , takes a snapshot S and computes $balance(a, S)$. If the amount to be transferred does not exceed $balance(a, S)$, we add the transfer operation to the set of p 's operations in the snapshot object via an *update* operation and return *true*. Otherwise, the operation returns *false*.

Theorem 1. *The asset-transfer object type has a wait-free implementation in the read-write shared memory model.*

Proof. Fix an execution E of the algorithm in Figure 1. Atomic snapshots can be wait-free implemented in the read-write shared memory model [AAD⁺93]. As every operation only involves a finite number of atomic snapshot accesses, every process completes each of the operations it invokes in a finite number of its own steps.

Let Ops be the set of:

- All invocations of *transfer* or *read* in E that returned, and
- All invocations of *transfer* in E that completed the *update* operation (line 5).

Let H be the history of E . We define a completion of H and, for each $o \in Ops$, we define a linearization point as follows:

Chapter 2. Asset Transfer in Shared Memory

- If o is a *read* operation, it linearizes at the linearization point of the *snapshot* operation in line 7.
- If o is a *transfer* operation that returns *false*, it linearizes at the linearization point of the *snapshot* operation in line 1.
- If o is a *transfer* operation that completed the *update* operation, it linearizes at the linearization point of the *update* operation in line 5. If o is incomplete in H , we complete it with response *true*.

Let \bar{H} be the resulting complete history and let L be the sequence of complete invocations of \bar{H} in the order of their linearization points in E . Note that, due to the way we linearize invocations, the linearization of a prefix of E is a prefix of L .

Now we show that L is legal and, thus, H is linearizable. We proceed by induction, starting with the empty (trivially legal) prefix of L . Let L_ℓ be the legal prefix of the first ℓ invocations and op be the $(\ell + 1)$ st operation of L . Let op be invoked by process p . The following cases are possible:

- op is a *read*(a): the snapshot taken at the linearization point of op contains all successful transfers concerning (incoming to or outgoing from) a in L_ℓ . By the induction hypothesis, the resulting balance is non-negative.
- op is a failed *transfer*(a, b, x): the snapshot taken at the linearization point of op contains all successful transfers concerning a in L_ℓ . By the induction hypothesis, the resulting balance is non-negative.
- op is a successful *transfer*(a, b, x): by the algorithm, before the linearization point of op , process p took a snapshot. Let L_k , $k \leq \ell$, be the prefix of L_ℓ that only contain operations linearized before the point in time when the snapshot was taken by p .

We observe that L_k includes a *subset* of all incoming transfers on a and *all* outgoing transfers on a in L_ℓ . Indeed, as p is the owner of a and only the owner of a can perform outgoing transfers on a , all outgoing transfers in L_ℓ were linearized before the moment p took the snapshot within op . Thus, $balance(a, L_k) \leq balance(a, L_\ell)$.²

By the algorithm, as $op = transfer(a, b, x)$ succeeds, we have $balance(a, L_k) \geq x$. Thus, $balance(a, L_\ell) \geq x$ and the resulting balance in $L_{\ell+1}$ is non-negative.

Thus, H is linearizable. □

Corollary 1. *The asset-transfer object type has consensus number 1.*

² Analogously to $balance(a, S)$ that computes the balance for account a based on the transfers contained in snapshot S , $balance(a, L)$, if L is a sequence of operations, computes the balance of account a based on all transfers in L .

2.4 k -Shared Asset-Transfer Has Consensus Number k

We now consider the case with an arbitrary owner map μ . We show that an asset-transfer object's consensus number is the maximal number of processes sharing an account. More precisely, the consensus number of an asset-transfer object is $\max_{a \in \mathcal{A}} (\mu(a))$.

We say that an asset-transfer object, defined on a set of accounts \mathcal{A} with an ownership map μ , is *k-shared* iff $\max_{a \in \mathcal{A}} (\mu(a)) = k$. In other words, the object is *k-shared* if μ allows at least one account to be owned by k processes, and no account is owned by more than k processes.

We show that the consensus number of any k -shared asset-transfer object is k , which generalizes our result in Corollary 1. We first show that such an object has consensus number *at least* k by implementing consensus for k processes using only registers and an instance of k -shared asset-transfer. We then show that k -shared asset-transfer has consensus number *at most* k by reducing it to k -consensus, an object known to have consensus number k [JT92].

Lemma 1. *Consensus has a wait-free implementation for k processes in the read-write shared memory model equipped with a single k -shared asset-transfer object.*

Algorithm 2 Wait-free implementation of consensus among k processes using a k -shared asset-transfer object and read-write registers. Code for process $p \in \{1, \dots, k\}$.

Shared variables:

- $R[i], i \in 1, \dots, k$, k registers, initially $R[i] = \perp, \forall i$
- AT , k -shared asset-transfer object containing:
 - an account a with initial balance $2k$
 - owned by processes $1, \dots, k$
 - some account s

Upon *propose*(v):

- 1 $R[p].write(v)$
 - 2 $AT.transfer(a, s, 2k - p)$
 - 3 **return** $R[AT.read(a)].read()$
-

Proof. The proof consists of providing a wait-free implementation of consensus in a shared memory system with k processes equipped with a k -shared asset transfer object. Intuitively, k processes use one shared account to elect one of them whose input value will be decided. Each process first announces its proposed value in a dedicated (per-process) register, and then tries to perform a transfer from the shared account. The process successfully performing a transfer is the elected one, and all processes use the remaining account balance to identify it.

Our algorithm is described in Algorithm 2. Before a process p accesses the shared account a , p announces its input in a register (line 1). Process p then tries to perform a transfer from account a to another account. The amount withdrawn this way from account a is chosen specifically such that:

Chapter 2. Asset Transfer in Shared Memory

1. only one transfer operation can ever succeed, and
2. if the transfer succeeds, the remaining balance on a will uniquely identify process p .

To satisfy the above conditions, we initialize the balance of account a to $2k$ and have each process $p \in \{1, \dots, k\}$ transfer $2k - p$ (line 2). Note that transfer operations invoked by distinct processes $p, q \in \{1, \dots, k\}$ have arguments $2k - p$ and $2k - q$, and $2k - p + 2k - q \geq 2k - k + 2k - (k - 1) = 2k + 1$. The initial balance of a is only $2k$ and no incoming transfers are ever executed. Therefore, the first transfer operation to be applied to the object succeeds (no transfer tries to withdraw more than $2k$) and the remaining operations will have to fail due to insufficient balance. When p reaches line 3, at least one transfer must have succeeded:

1. either p 's transfer succeeded, or
2. p 's transfer failed due to insufficient balance, in which case some other process must have previously succeeded.

Let q be the process whose transfer succeeded. Thus, the balance of account a is $2k - (2k - q) = q$. Since q performed a transfer operation, by the algorithm, q must have previously written its proposal to the register $R[q]$. Regardless of whether $p = q$ or $p \neq q$, reading the balance of account a returns q and p decides the value of $R[q]$. \square

To prove that k -shared asset-transfer has consensus number at most k , we reduce k -shared asset-transfer to k -consensus. A k -consensus object exports a single operation *propose* that, the first k times it is invoked in a history, returns the argument of the first invocation. All subsequent invocations return \perp . Given that k -consensus is known to have consensus number exactly k [JT92], a wait-free algorithm implementing k -shared asset-transfer using only registers and k -consensus objects implies that the consensus number of k -shared asset-transfer is not more than k .

The algorithm reducing k -shared asset-transfer to k -consensus is given in Algorithm 3 (with auxiliary functions defined in Algorithm 4). It is a generalization of the single-owner case (Algorithm 1). It follows a scheme similar to the universal construction of non-deterministic objects [Her91], where processes explicitly agree not only on the order of the performed operations, but also on their results (success or failure).

Before presenting a formal correctness argument, we first informally explain the intuition behind the reduction. In our algorithm, we associate a series of k -consensus objects with every account a . Up to k owners of a use the k -consensus objects to agree on the order of outgoing transfers for a . Similarly to the single-owner case (Algorithm 1), the processes publish these transfers in an atomic snapshot object AS . Every process p uses a distinct entry of AS to store a set of transfers outgoing from accounts p owns.

Algorithm 3 Wait-free implementation of a k -shared asset-transfer object using k -consensus objects. Code for process p .

Shared variables:

AS , atomic snapshot object
for each $a \in \mathcal{A}$:
 $R_a[i], i \in \Pi$, registers, initially $[\perp, \dots, \perp]$
 $kC_a[i], i \geq 0$, list of instances of k -consensus objects

Local variables:

$hist$: a set of completed transfers, initially empty
for each $a \in \mathcal{A}$:
 $committed_a$, initially \emptyset
 $round_a$, initially 0

Upon $transfer(a, b, x)$:

```

1  if  $p \notin \mu(a)$  then
2    return false
3   $tx = (a, b, x, p, round_a)$ 
4   $R_a[p].write(tx)$ 
5   $collected = collect(a) \setminus committed_a$ 
6  while  $tx \in collected$  do
7     $req =$  the oldest transfer in  $collected$ 
8     $prop = proposal(req, AS.snapshot())$ 
9     $decision = kC_a[round_a].propose(prop)$ 
10    $hist = hist \cup \{decision\}$ 
11    $AS.update(hist)$ 
12    $committed_a = committed_a \cup \{t : decision = (t, *)\}$ 
13    $collected = collected \setminus committed_a$ 
14    $round_a = round_a + 1$ 
15  if  $(tx, success) \in hist$  then
16    return true
17  else
18    return false

```

Upon $read(a)$:

```

19  return  $balance(a, AS.snapshot())$ 

```

Chapter 2. Asset Transfer in Shared Memory

Algorithm 4 Auxiliary functions used by algorithm in Algorithm 3

```
collect(a):
1  collected =  $\emptyset$ 
2  for all  $i = \Pi$  do
3    if  $R_a[i].read() \neq \perp$  then
4       $collected = collected \cup \{R_a[i].read()\}$ 
5  return collected

proposal((a, b, q, x), snapshot):
6  if  $balance(a, snapshot) \geq x$  then
7     $prop = ((a, b, q, x), success)$ 
8  else
9     $prop = ((a, b, q, x), failure)$ 
10 return prop

balance(a, snapshot):
11  $incoming = \{tx : tx = (*, a, *, *, *) \wedge (tx, success) \in snapshot\}$ 
12  $outgoing = \{tx : tx = (a, *, *, *, *) \wedge (tx, success) \in snapshot\}$ 
13 return  $q_0(a) + (\sum_{(*, a, *, *, *) \in incoming} x) - (\sum_{(a, *, *, *, *) \in outgoing} x)$ 
```

However, unlike in the single-owner case, establishing a (per-account) total order of outgoing transfers is not sufficient to ensure linearizability of k -shared asset-transfer. In the single-owner case, a process only appends valid transfers (those with sufficient funds) to its account history (and thus each transfer found in the history can be considered successful). This is possible, because a single owner p can first check whether a transfer is valid (by taking a snapshot and checking the balance) and, if so, append the transfer to the published history of operations on the account (otherwise returning false and forgetting that transfer forever). This, in turn, is possible, because, in the single-owner case, no other process can decrease the balance of the account between the moment when p checks the validity of the outgoing transfer and the moment it appends the transfer to the account history. For shared accounts, this no longer holds, as co-owners might be concurrently proposing conflicting transfers.

Intuitively, even if the processes sharing a agree the same order of outgoing transfers, they may still observe a different subset of incoming transfers. In case of concurrent incoming transfers to a , processes sharing a might observe different balances of a when evaluating the success / failure of an outgoing transfer. Note that this is not an issue in the case of a single account per process, where a process only appends successful transfers (those with sufficient funds) to its account history.

Therefore, every process p stores in its entry of AS a set *hist*—a subset of all completed outgoing transfers from accounts that p owns (and thus is allowed to debit) and their results. Each element in the *hist* set is represented as $((a, b, x, s, r), result)$, where a, b , and x are the respective source account, destination account, and the amount transferred, s is the originator

2.4. k -Shared Asset-Transfer Has Consensus Number k

of the transfer, and r is the *round* in which the transfer was invoked by the originator. The value of $result \in \{\text{success}, \text{failure}\}$ indicates whether the transfer succeeds or fails. A transfer becomes “visible” when any process inserts it in its corresponding entry of AS .

To read the balance of account a , a process takes a snapshot of AS , and then sums the initial balance $q_0(a)$ and amounts of all successful incoming transfers, and subtracts the amounts of successful outgoing transfers found in AS . We say that a successful transfer tx is in a snapshot AS (denoted by $(tx, \text{success}) \in AS$) if there exists an entry e in AS such that $(tx, \text{success}) \in AS[e]$.

To execute a transfer o outgoing from account a , a process p first announces o in a register R_a that can be written by p and read by any other process (line 4). This enables a “helping” mechanism needed to ensure wait-freedom to the owners of a [Her91].

Next, p collects the transfers proposed by other owners (line 5) and tries to agree on the order of the collected transfers and their results using a series of k -consensus objects. For each account, the agreement on the order of transfer-result pairs proceeds in rounds. Each round is associated with a k -consensus object which p invokes with a proposal chosen from the set of collected transfers (line 9). Since each process, in each round, only invokes the k -consensus object once, no k -consensus object is invoked more than k times and thus each invocation returns a value (and not \perp).

A transfer-result pair as a proposal for the next instance of k -consensus is chosen as follows. Process p picks the “oldest” collected but not yet committed operation (based on the round number $round_a$ attached to the transfer operation when a process announces it; ties are broken using process IDs) (line 7). Then p takes a snapshot of AS and checks whether account a has sufficient balance according to the state represented by the snapshot, and equips the transfer with a corresponding *success* / *failure* flag (line 8). The resulting transfer-result pair constitutes p ’s proposal for the next instance of k -consensus (line 9).

For each round, p inserts the decided-upon transfer-result pair in its *hist* set (line 10). p keeps executing rounds of k -consensus until its outgoing transfer o has been decided in some round. Locally, p keeps track of all transfers that have been agreed upon so far and excludes them from the set of collected transfers before constructing its proposal in each round.

The currently executed transfer by process p returns as soon as it is decided by a k -consensus object, the flag of the decided value (*success* / *failure*) indicating the transfer’s response (*true* / *false*). Note that every process that keeps proposing transfers to the k -consensus objects will eventually learn about o . Moreover, o will eventually become the oldest announced transfer and thus will be decided in some round. Therefore, p ’s *transfer* operation will always eventually terminate.

Lemma 2. *The k -shared asset-transfer object type has a wait-free implementation in the read-write shared memory model equipped with k -consensus objects.*

Chapter 2. Asset Transfer in Shared Memory

Proof. We essentially follow the footpath of the proof of Theorem 1. Fix an execution E of the algorithm in Figure 3. Let H be the history of E .

To perform a transfer o on an account a , p registers it in $R_a[p]$ (line 4) and then proceeds through a series of k -consensus objects, each time collecting R_a to learn about the transfers concurrently proposed by other owners of a . Recall that each k -consensus object is wait-free. Suppose, by contradiction, that o is registered in R_a but is never decided by any instance of k -consensus. Eventually, however, o becomes the request with the lowest round number in R_a and, thus, some instance of k -consensus will be only accessed with o as a proposed value (line 9). By validity of k -consensus, this instance will return o and, thus, p will be able to complete o .

Let Ops be the set of all complete operations and all *transfer* operations o such that some process completed the update operation (line 11) in E with an argument including o (the atomic snapshot and k -consensus operation has been linearized). Intuitively, we include in Ops all operations that *took effect*, either by returning a response to the user or by affecting other operations. Recall that every such *transfer* operation was agreed upon in an instance of k -consensus, let it be kC^o . Therefore, for every such *transfer* operation o , we can identify the process q^o whose proposal has been decided in that instance.

We now determine a completion of H and, for each $o \in Ops$, we define a linearization point as follows:

- If o is a *read* operation, it linearizes at the linearization point of the snapshot operation (line 19).
- If o is a *transfer* operation that returns *false*, it linearizes at the linearization point of the snapshot operation (line 8) performed by q^o just before it invoked $kC^o.propose()$.
- If o is a *transfer* operation that some process included in the update operation (line 11), it linearizes at the linearization point of the *first* update operation in H (line 11) that includes o . Furthermore, if o is incomplete in H , we complete it with response *true*.

Let \tilde{H} be the resulting complete history and let L be the sequence of complete operations of \tilde{H} in the order of their linearization points in E . Note that, by the way we linearize operations, the linearization of a prefix of E is a prefix of L . Also, by construction, the linearization point of an operation belongs to its interval.

Now we show that L is legal and, thus, H is linearizable. We proceed by induction, starting with the empty (trivially legal) prefix of L . Let L_ℓ be the legal prefix of the first ℓ operation and op be the $(\ell + 1)$ st operation of L . Let op be invoked by process p . The following cases are possible:

- op is a *read*(a): the snapshot taken at op 's linearization point contains all successful

transfers concerning a in L_ℓ . By the induction hypothesis, the resulting balance is non-negative.

- op is a failed $transfer(a, b, x)$: the snapshot taken at the linearization point of op contains all successful transfers concerning a in L_ℓ . By the induction hypothesis, the balance corresponding to this snapshot non-negative. By the algorithm, the balance is less than x .
- op is a successful $transfer(a, b, x)$. Let L_s , $s \leq \ell$, be the prefix of L_ℓ that only contains operations linearized before the moment of time when q^o has taken the snapshot just before accessing kC^o .

As before accessing kC^o , q went through all preceding k -consensus objects associated with a and put the decided values in AS , L_s must include *all* outgoing $transfer$ operations for a . Furthermore, L_s includes a *subset* of all incoming transfers on a . Thus, $balance(a, L_k) \leq balance(a, L_\ell)$.

By the algorithm, as $op = transfer(a, b, x)$ succeeds, we have $balance(a, L_k) \geq x$. Thus, $balance(a, L_\ell) \geq x$ and the resulting balance in $L_{\ell+1}$ is non-negative.

Thus, H is linearizable. □

Theorem 2. *A k -shared asset-transfer object has consensus number k .*

Proof. It follows directly from Lemma 1 that k -shared asset-transfer has consensus number at least k . Moreover, it follows from Lemma 2 that k -shared asset-transfer has consensus number at most k . Thus, the consensus number of k -shared asset-transfer is exactly k . □

3 Asset Transfer in Message Passing

In the previous chapter we studied the problem of asset transfer as a concurrent object type in the shared memory model with crash-only failures. We proved that with consensus number 1, such a concurrent object occupies the lowest rank of Herlihy's consensus hierarchy [Her91] and, consequently, does not require solving the consensus problem in order to be implemented.

In this chapter, we provide analogous results in the message passing model in presence of a Byzantine adversary. We implement asset transfer without having to solve distributed consensus, and we provide an intuition on how to generalize our algorithm to the k -shared case, where k processes sharing an account can solve agreement among themselves.

3.1 Introduction

The shared memory model is well suited for precisely studying theoretical aspects of many problems and it served us well to understand the relation between asset transfer and consensus. In practice, it is most useful for describing concurrent processes accessing a shared physical memory. These processes are mostly either running on different CPU cores of the same machine, or are distributed across a fast network (usually within a datacenter) where the exchanged messages are abstracted away and each process indeed deals with an abstraction of a shared memory.

However, we envision a practical implementation of asset transfer to be deployed as a large-scale distributed system spanning all across the planet. In such a setting, the message passing model, where processes communicate by exchanging messages, is better suited. While an abstraction of shared memory can be (under some additional assumptions) implemented on top of message passing [ABND95], such an implementation is complex, expensive, and has limited scalability.

Moreover, our goal is to also achieve Byzantine fault tolerance. While a shared memory abstraction can be implemented in a message passing system even with the presence of

Chapter 3. Asset Transfer in Message Passing

Byzantine processes [GV06, GLV06, GV07, IRRS16], our algorithms presented in the previous chapter do not directly translate to the Byzantine environment.

We now present a practical implementation of asset transfer in the message passing model that is tolerant to Byzantine faults. Instead of relying on a shared memory abstraction and an atomic snapshot object to represent the system state, every process maintains its own representation of the accounts and their balances. Processes communicate using a secure broadcast primitive that they use to disseminate information about the performed transfers.

An important aspect of our approach is that we still avoid solving the consensus problem. The secure broadcast primitive we use need not provide a total order among messages (containing transfers in our case). A much weaker per-process ordering, called *source order* [MR97b], is sufficient. This order only concerns messages that have been broadcast by the same process.

We leverage the insight that only outgoing transfers from an account need to be totally ordered with respect to each other. In particular, outgoing transfers from one account do not need to be ordered with respect to transfers from another account. Since incoming transfers for an account are commutative, a process can observe them in any order. The only ordering constraint we enforce is a weak form of causality. This prevents situations where a process, unaware of an account having received an asset, would observe this asset being further transferred from this account. Such a weak ordering is, unlike total order, easily achievable asynchronously.

Analogously to the previous chapter, we discuss a generalization to the k -shared case. We describe how to adapt our algorithm to implement k -shared asset transfer, under the assumption that up to k processes are able to solve Byzantine agreement among themselves.

The rest of this chapter is organized as follows. First, we adapt our specification of asset transfer to the message passing environment (3.2) with Byzantine faults. Next, we present our consensusless algorithm implementing asset transfer and prove its correctness (3.3). Finally, we discuss the adaptations that are necessary to generalize our implementation to the k -shared case (3.4).

3.2 Byzantine-Tolerant Asset Transfer

We now adapt our asset transfer specification for an environment where processes communicate by sending messages over reliable authenticated channels [CGR11], and where up to one third of the processes can be subject to *Byzantine* failures.

A process is Byzantine if it deviates from the algorithm it is assigned, either by halting prematurely, in which case we say that the process is *crashed*, or performing actions that are not prescribed by its algorithm, in which case we say that the process is *malicious*. Malicious processes can perform arbitrary actions, except for ones that involve subverting cryptographic primitives. A process is called *faulty* if it is either crashed or malicious. A process is *correct* if it

3.3. Asset Transfer Implementation in Byzantine Message Passing

is not faulty and *benign* if it is not malicious. Note that every correct process is benign, but not necessarily vice versa.

We only require that the transfer system behaves correctly towards *benign* processes, regardless of the behavior of Byzantine ones. Informally, we require that no benign process can be a victim of a double-spending attack, i.e., every execution appears to benign processes as a correct sequential execution, respecting the original execution's real-time ordering [Her91].

In our algorithm, we slightly relax the last requirement—while still preventing double-spending. We require that *successful transfer* operations invoked by benign processes constitute a legal sequential history that preserves the real-time order. A *read* or a failed *transfer* operation invoked by a benign process p can be “outdated”—it can be based on a stale state of p 's balance. Informally, one can view the system requirements as *linearizability* [HW90] for successful transfers and *local sequential consistency* [AW94] for failed transfers and reads. One can argue that this relaxation incurs little impact on the system's utility, since all incoming transfers are eventually applied. As progress (liveness) guarantees, we require that every operation invoked by a correct process eventually completes.

Definition 1 (Correctness of asset-transfer in message passing). *Let E be any execution of an implementation and H be the corresponding history. Let $ops(H)$ denote the set of operations in H that were executed by correct processes in E . An asset-transfer object in message passing guarantees that each invocation issued by a correct process is followed by a matching response in H , and that there exists \bar{H} , a completion of H , such that:*

- (1) *Let \bar{H}^t denote the sub-history of successful transfers of \bar{H} performed by correct processes and $<_{\bar{H}}^t$ be the subset of $<_{\bar{H}}$ restricted to operations in \bar{H}^t . Then there exists a legal sequential history S such that (a) for every correct process p , $\bar{H}^t|p = S|p$ and (b) $<_{\bar{H}}^t \subseteq <_S$.*
- (2) *For every correct process p , there exists a legal sequential history S_p such that:*

- $ops(\bar{H}) \subseteq ops(S_p)$, and
- $S_p|p = \bar{H}|p$.

Notice that property (2) implies that every update in H that affects the account of a correct process p is eventually included in p 's “local” history and, therefore, will reflect reads and transfer operations subsequently performed by p .

3.3 Asset Transfer Implementation in Byzantine Message Passing

We now present our consensusless algorithm that implements asset transfer in a message passing system subject to Byzantine faults. Instead of consensus, we rely on a secure broadcast primitive (referred to as “secure reliable multicast” by Malkhi et al. [MMR97, MR97b]) that is strictly weaker than consensus and has a fully asynchronous implementation. It provides

reliable delivery despite Byzantine faults and so-called *source order* among delivered messages. The source order property, being even weaker than FIFO, guarantees that messages from the same source are delivered in the same order by all correct processes. More precisely, the secure broadcast primitive we use in our implementation (with a standard broadcast/delivery interface) has the following properties for processes p , q , and r , and messages m and m' (paraphrasing from [MR97b]):

- **Integrity:** A benign process delivers a message m from a process p at most once and, if p is benign, only if p previously broadcast m .
- **Agreement:** If processes p and q are correct and p delivers m , then q delivers m .
- **Validity:** If a correct process p broadcasts m , then p delivers m .
- **Source order:** If p and q are benign and both deliver m from r and m' from r , then they do so in the same order.

Figure 3.1 depicts the high-level structure of our algorithm. There are two main modules: one for tracking dependencies among transfer operations, and one module for securely broadcasting new transfers.

The secure broadcast module is a classic broadcast abstraction with the properties described above that has known implementations [MR97b, MMR97]. In the next chapter we present a probabilistic version of secure broadcast along with a new implementation that vastly improves scalability for a cost of an (arbitrarily small) failure probability.

Now we focus on the asset transfer algorithm that uses a secure broadcast abstraction as a black box. This algorithm remains the same regardless of the underlying variant of secure broadcast. If using a deterministic implementation of secure broadcast, our algorithm provides deterministic guarantees. In the other case, we inherit the probabilistic behavior and the asset transfer algorithm will also be allowed to fail with a certain (still arbitrarily small) probability.

The intuition behind our algorithm is as follows. To perform a transfer tx , a process p securely broadcasts a message with the transfer details: the arguments of the *transfer* operation and some metadata. The metadata includes a per-process *sequence number* of tx and references to the *dependencies* of tx . The dependencies are transfers incoming to p that must be known to any process before applying tx .¹ These dependencies impose a causal relation between transfers that must be respected when transfers are being applied. For example, suppose that process p makes a transfer tx to process q . q , after observing tx , performs another transfer tx' to process r . q 's broadcast message will contain tx' , a local sequence number, and a reference to tx . Any process (not only r) will only evaluate the validity of tx' after having applied tx . This approach is similar to using vector clocks for implementing causal order among events [JF88].

¹ By applying a transfer tx we understand locally adding tx to a set of transfers considered as executed.

3.3. Asset Transfer Implementation in Byzantine Message Passing

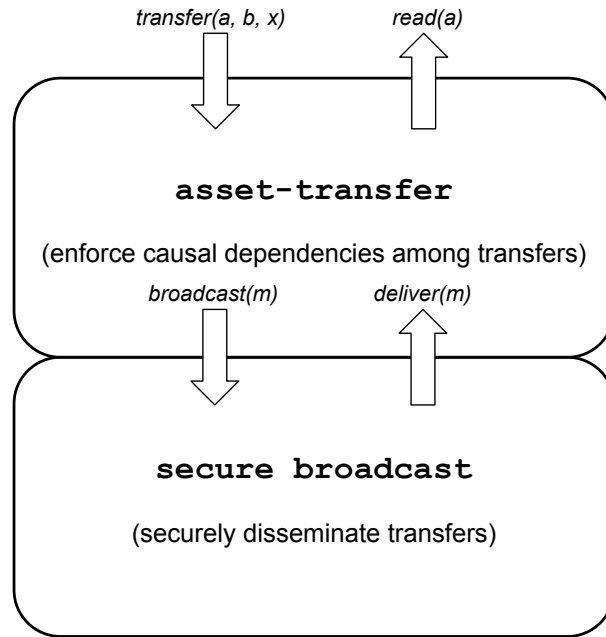


Figure 3.1 – High-level structure of our message-passing implementation of asset transfer. The upper layer (whose algorithm is depicted in Algorithm 5) uses a secure broadcast abstraction as a black box.

To ensure the authenticity of operations—so that no process is able to debit another process’s account—we assume that processes sign all their messages before broadcasting them. In practice, similar to Bitcoin and other transfer systems, every process p possesses a public-private key pair that allows only p to securely initiate transfers from its corresponding account. For simplicity of presentation, we omit this mechanism in the algorithm pseudocode.

Algorithm 5 describes the full algorithm implementing asset-transfer in a Byzantine-prone message passing system. Each process p maintains, for each process q , an integer $seq[q]$ reflecting the number of transfers which process q initiated and which process p has validated and applied. Process p also maintains, for every process q , an integer $rec[q]$ which reflects the number of transfers which process q has initiated and process p has delivered (but not necessarily applied).

Additionally, there is also a set $hist[q]$ of transfers which *involve* process q . We say that, intuitively, a transfer operation involves a process q if that transfer is either outgoing or incoming on the account of q . However, we exclude from this set the transfers incoming for q that have not yet been referenced by any of q ’s outgoing transfers. Thus, a transfer tx from r to q is inserted into $hist[r]$ immediately after validation (as it is outgoing from r), but we only insert it into $hist[q]$ when q issues another transfer referencing tx as a dependency. This small technical detail is important for evaluating the validity of transfers issued by (and thus outgoing from) q . Consider the case where q is malicious and issues a transfer tx that is not justified by the referenced dependencies. Consequently, a correct process p will not validate

Chapter 3. Asset Transfer in Message Passing

Algorithm 5 Consensusless transfer system based on secure broadcast. Code for process p .

Local variables:

$seq[]$, initially $seq[q] = 0, \forall q$ {Number of validated transfers outgoing from q }

$rec[]$, initially $rec[q] = 0, \forall q$ {Number of delivered transfers from q }

$hist[]$, initially $hist[q] = \emptyset, \forall q$ {Set of validated transfers outgoing from q and their dependencies}

$deps$, initially \emptyset {Set of last incoming transfers for account of local process p }

$toValidate$, initially \emptyset {Set of delivered (but not validated) transfers}

$q_0[]$, initially $q_0[q] = \text{initial balance of } q, \forall q$ {Initial account balances}

```
1 operation  $transfer(a, b, x)$  where  $\mu(a) = \{p\}$            { Transfer an amount of  $x$  from account  $a$  to account  $b$  }
2   if  $balance(a, hist[p] \cup deps) < x$  then
3     return  $false$ 
4      $broadcast([(a, b, x, seq[p] + 1), deps])$ 
5      $deps = \emptyset$ 

6 operation  $read(a)$            { Read balance of account  $a$  }
7   return  $balance(a, \bigcup_{q \in \Pi} hist[q])$ 

{ Secure broadcast delivery }
8 upon  $deliver(q, m)$            { Executed when  $p$  delivers message  $m$  from process  $q$  }
9   let  $m$  be  $[(q, b, x, s), deps]$ 
10  if  $s = rec[q] + 1$  then
11     $rec[q] = rec[q] + 1$ 
12     $toValidate = toValidate \cup \{(q, m)\}$ 

{ Executed when a transfer delivered from  $q$  becomes valid }
13 upon  $(q, [t, h]) \in toValidate \wedge Valid(q, t, h)$ 
14    $hist[q] := hist[q] \cup h \cup \{t\}$            { Update the history for the outgoing account }
15   let  $t$  be  $(q, b, x, s)$ 
16    $seq[q] = s$ 
17   if  $b = p$  then
18      $deps = deps \cup \{(q, b, x, s)\}$            { This transfer is incoming to account of local process  $p$  }
19   if  $q = p$  then
20     return  $true$            { This transfer is outgoing from account of local process  $p$  }

21 function  $Valid(q, t, h)$ 
22   let  $t$  be  $(c, d, y, s)$ 
23   return  $(q = c)$ 
24     and  $(s = seq[q] + 1)$ 
25     and  $(balance(c, hist[q] \cup h) \geq y)$ 
26     and  $(forall(a, b, x, r) \in h : (a, b, x, r) \in hist[a])$ 

27 function  $balance(a, h)$ 
28   return initial balance  $q_0[a]$  plus sum of incoming transfers minus outgoing transfers for account  $a$  in  $h$ 
```

3.3. Asset Transfer Implementation in Byzantine Message Passing

it. If processes were allowed to take into account unreferenced incoming transfers, another correct process r might validate tx due to additional incoming transfers p is not yet aware of.

Each process p maintains as well a local variable $deps$. This is a set of transfers *incoming* for p that p has applied since the last successful *outgoing* transfer. Finally, the set $toValidate$ contains delivered transfers that have been delivered from the underlying broadcast, but not yet applied.

To perform a *transfer* operation, process p first checks the balance of its own account, and if the balance is insufficient, returns *false* (line 3). Otherwise, process p broadcasts a message with this operation via the secure broadcast primitive (line 4). This message includes the three basic arguments of a *transfer* operation as well as $seq[p] + 1$ and dependencies $deps$. Each correct process in the system eventually delivers this message via secure broadcast (line 8). Note that, given the assumption of no process executing more than one concurrent transfer, every process waits for delivery of its own message before initiating another broadcast. This effectively turns the source order property of secure broadcast into FIFO order. Upon delivery, process p checks this message for well-formedness (lines 9 and 10), and then adds it to the set of messages pending validation. We explain the validation procedure later.

Once a transfer passes validation (the predicate in line 13 is satisfied), process p applies this transfer on the local state. Applying a transfer means that process p adds this transfer along with its dependencies to the history of the outgoing account (line 14). If the transfer is incoming for local process p , it is also added to $deps$, the set of current dependencies for p (line 18). If the transfer is outgoing for p , i.e., it is the currently pending *transfer* operation invoked by p , then the response *true* is returned (line 20).

To perform a *read(a)* operation for account a , process p simply computes the balance of this account based on the union of all currently validated transfers (line 7).

Before applying a transfer op from some process q , process p validates op via the *Valid* function (lines 21–26). To be valid, op must satisfy four conditions. The first condition is that process q (the issuer of transfer op) must be the owner of the outgoing account for op (line 23). Second, any preceding transfers that process q issued must have been validated (line 24). Third, the balance of account q must not drop below zero (line 25). Finally, every dependency (a, b, x, r) in h must have been validated and included in the history of a , $hist[a]$.

Lemma 3. *In any infinite execution of the algorithm (Figure 5), every operation performed by a correct process eventually completes.*

Proof. A transfer operation that fails or a read operation invoked by a correct process returns immediately (lines 3 and 7, respectively).

Consider a transfer operation tx invoked by a correct process p that *succeeds* (i.e., passes the check in line 2), so p broadcasts a message with the transfer details using secure broadcast (line 4). By the validity property of secure broadcast, p eventually delivers the message (via

the secure broadcast callback, line 8) and adds it to the *toValidate* set. By the algorithm, this message includes a set *deps* of operations (called *h*, line 9) that involve *p*'s account. This set includes transfers that process *p* delivered and validated after issuing the prior successful outgoing transfer (or since the initial system time if there is no such transfer) but before issuing *tx* (lines 4 and 5).

As process *p* is correct, it operates on its own account, respects the sequence numbers, and issues a transfer only if it has enough balance on the account. Thus, when a transfer operation *tx* is delivered by *p*, it must satisfy the first three conditions of the *Valid* predicate (lines 23–25).

Consider any (a, b, x, r) in *h*, the dependency set attached by *p* to the broadcast message (line 4) and then used as a parameter in function *Valid* (line 4). By the algorithm, before being included to the dependency set, (a, b, x, r) must be first validated and added to *hist*[*a*] (line 14). Thus, the fourth validation condition (line 26) also holds.

Hence, *p* eventually validates *tx* and completes the operation by returning *true* in line 20. \square

Theorem 3. *The algorithm in Figure 5 implements an asset-transfer object type.*

Proof. Fix an execution *E* of the algorithm, let *H* be the corresponding history.

Let \mathcal{V} denote the set of all messages that were delivered (line 8) and validated (line 23) at a correct process in *E*. By the agreement property of secure broadcast, every message $m = [(q, d, y, s), h] \in \mathcal{V}$ is eventually put in *hist*[*q*] (line 14) at each correct process. Now we define an order $\leq \subseteq \mathcal{V} \times \mathcal{V}$ as follows. For $m = [(q, d, y, s), h] \in \mathcal{V}$ and $m' = [(r, d', y', s'), h'] \in \mathcal{V}$, we have $m \leq m'$ if and only if one of the following conditions holds:

- $q = r$ and $s < s'$,
- $(r, d, y, s) \in h'$, or
- there exists $m'' \in \mathcal{V}$ such that $m \leq m''$ and $m'' \leq m'$.

By the source order property of secure broadcast (see Section 3.3), correct processes *p* and *r* deliver messages from any process *q* in the same order. By the algorithm in Algorithm 5, a message from *q* with a sequence number *i* is added by a correct process to *toValidate* set only if the previous message from *q* added to *toValidate* had sequence number *i* – 1 (line 10). Furthermore, a message $m = [(q, d, y, s), h]$ is validated at a correct process only if all messages in *h* have been previously validated (line 26). Therefore, \leq is acyclic and can be extended to a total order.

Let *S* be the sequential history constructed from any such total order on messages in \mathcal{V} in which every message $m = [(q, d, y, s), h]$ is replaced with the invocation-response pair *transfer*(*q*, *d*, *y*); *true*.

3.4. k -shared Asset Transfer in Message Passing

By the definition of the order \leq , every operation $transfer(q, d, y)$ in S is preceded by a sequence of transfers ensuring that the balance of q does not drop below y (line 25). In particular, S includes all outgoing transfers from the account of q performed previously by q itself. Additionally S may order some *incoming* transfer to q that did not appear at $hist[q]$ before the corresponding (q, d, y, s) has been added to it. But these “unaccounted” operations may only increase the balance of q and, thus, it is indeed legal to return *true*.

Furthermore, for each correct process p , \leq and, thus, S respects the order of successful transfers issued by p . Thus, the subsequence of successful transfers in H “looks” linearizable to the correct processes: H , restricted to successful transfers witnessed by the correct processes, is consistent with a legal sequential history S .

Let p be a correct process in E . Now let \mathcal{V}_p denote the set of all messages that were delivered (line 8) and validated (line 23) at p in E . Let \leq_p be the subset of \leq restricted to the elements in \mathcal{V}_p . Obviously, \leq_p is cycle-free and we can again extend it to a total order. Let S_p be the sequential history built in the same way as S above. Similarly, we can see that S_p is legal and, by construction, consistent with the local history of *all* operations of p (including reads and failed transfers). Moreover, every successful transfer in E performed by a correct process eventually appears in $\bigcup_{q \in \Pi} hist[q]$. Thus, every such transfer is eventually included in S_p .

By Lemma 3, every operation invoked by a correct process eventually completes. Thus, E indeed satisfies the requirement of an asset-transfer implementation. \square

3.4 k -shared Asset Transfer in Message Passing

Our message-passing asset-transfer implementation can be naturally extended to the k -shared case, when some accounts are owned by up to k processes. As we showed in Section 2.4, a purely asynchronous implementation of a k -shared asset-transfer does not exist, even in the benign shared-memory environment.

k -shared BFT service. To circumvent this impossibility, we assume that every account is associated with a Byzantine fault-tolerant state-machine replication service (BFT [CL99]) that is used by the account’s owners to order their outgoing transfers. More precisely, the transfers issued by the owners are assigned monotonically increasing *sequence numbers*.

The service can be implemented by the owners themselves, acting both as *clients*, submitting requests, and *replicas*, reaching agreement on the order in which the requests must be served. As long as more than two thirds of the owners are correct, the service is *safe*, in particular, no sequence number is assigned to more than one transfer. Moreover, under the condition that the owners can eventually communicate within a bounded message delay, every request submitted by a correct owner is guaranteed to be eventually assigned a sequence number [CL99]. One can argue that it is much more likely that this assumption of *eventual synchrony* holds for a bounded set of owners, rather than for the whole set of system participants. Furthermore,

communication complexity of such an implementation is polynomial in k and not in N , the number of processes.

Account order in secure broadcast. Consider even the case where the threshold of one third of Byzantine owners is exceeded, where the account may become blocked or, even worse, compromised. In this case, different owners may be able to issue two different transfers associated with the same sequence number.

This issue can be mitigated by a slight modification of the classical secure broadcast algorithm [MR97b]. In addition to the properties of Integrity, Validity and Agreement of secure broadcast, the modified algorithm can implement the property of *account order*, generalizing the *source order* property (Section 3.3). Assume that each broadcast message is equipped with a sequence number (generated by the BFT service, as we will see below).

- **Account order:** If a benign process p delivers messages m (with sequence number s) and m' (with sequence number s') such that m and m' are associated with the same account and $s < s'$, then p delivers m before m' .

Informally, the implementation works as follows. The sender sends the message (containing the account reference and the sequence number) it wants to broadcast to all and waits until it receives acknowledgements from a *quorum* of more than two thirds of the processes. A message with a sequence number s associated with an account a is only acknowledged by a benign process if the last message associated with a it delivered had sequence number $s - 1$. Once a quorum is collected, the sender sends the message equipped with the signed quorum to all and delivers the message. This way, the benign processes deliver the messages associated with the same account in the same order. If the owners of an account send conflicting messages for the same sequence number, the account may block. However, and most importantly, even a compromised account is always prevented from double spending. Liveness of operations on a compromised account is not guaranteed, but safety and liveness of other operations remains unaffected.

Putting it all together. The resulting k -shared asset transfer system is a composition of a collection of BFT services (one per account), the modified secure broadcast protocol (providing the account-order property), and a slightly modified protocol in Figure 5.

To issue a transfer operation t on an account a it owns, a process p first submits t to the associated BFT service to get a sequence number. Assuming that the account is not compromised and the service is consistent, the transfer receives a unique sequence number s . Note that the decided tuple (a, t, s) should be signed by a quorum of owners: this will be used by the other processes in the system to ensure that the sequence number has been indeed agreed upon by the owners of a . The process executes the protocol in Figure 5, with the only modification that the sequence number seq is now not computed locally but adopted from the BFT service.

Intuitively, as the transfers associated with a given account are processed by the benign

3.4. *k*-shared Asset Transfer in Message Passing

processes in the same order, the resulting protocol ensures that the history of successful transfers is linearizable. On the liveness side, the protocol ensures that every transfer on a non-compromised account is guaranteed to complete.

4 Probabilistic Secure Broadcast

Secure broadcast is a powerful primitive that allows a set of processes to agree on a message from a designated sender, even if some processes are Byzantine. The asset transfer algorithm described in the previous chapter uses secure broadcast as a black-box component as the only means of communication between processes. In fact, from the point of view of a process, the asset transfer algorithm can be seen as a relatively thin wrapper of local computation around secure broadcast. The implementation of this sub-protocol thus has a strong impact on the complexity and performance of our asset transfer algorithm.

Classic deterministic secure broadcast protocols build on *quorum systems* providing intersection guarantees, which results in linear per-process communication and computation complexity, severely limiting the scalability of these protocols.

In this chapter we generalize the secure broadcast abstraction to the probabilistic setting, allowing each property to be violated with a fixed, arbitrarily small probability. We leverage these relaxed guarantees in a protocol where we replace quorums with stochastic *samples*. Compared to quorums, samples are significantly smaller in size, leading to a more scalable design. We obtain the first secure broadcast protocol with logarithmic latency, as well as logarithmic per-process communication and computation complexity.

4.1 Introduction

Broadcast is a popular abstraction in the distributed system toolbox, allowing a process to transmit messages to a set of processes. Multiple flavors of broadcast have been defined in the literature, with different safety and liveness guarantees [CGR11, GKKZ11, HT93, MMR97, PS02]. In this thesis we focus on broadcast that tolerates Byzantine faults. Such broadcast abstractions are often central building blocks of practical Byzantine fault-tolerant (BFT) systems in general [CP02, DRZ18], and our asset transfer system in particular. This chapter tackles the problem of scalability, namely reducing the complexity of secure broadcast, and seeking good performance despite a large number of participating processes.

The broadcast abstraction mainly discussed in this chapter is slightly different from the one used to implement asset transfer in Section 3.3. In a nutshell, while our asset transfer algorithm uses a “multi-shot” version of secure broadcast, where any process may invoke the broadcast any number of times, we now restrict ourselves to a “single-shot” version, where only a single process is broadcasting a single message. However, the multi-shot variant of secure broadcast can be trivially implemented by instantiating multiple instances of the single-shot variant and using sequence numbers to implement source order. Unless stated otherwise, we refer to the single-shot version of secure broadcast in this chapter.

In single-shot secure broadcast, a designated *sender* can broadcast a single message, ensuring that no two correct processes deliver different messages. This holds despite the presence of a certain fraction of Byzantine processes, including the sender. Secure broadcast ensures three guarantees: (1) *Validity*: if the sender is correct, then its message is delivered by all correct processes; (2) *Consistency*: all correct processes that deliver a message should deliver the same message; (3) *Totality*: either every correct process delivers a message, or no correct process does so.

We denote by N the number of processes in the system, and f the fraction of processes that are Byzantine. Existing algorithms for secure broadcast typically have $O(N)$ per-process communication complexity [BT85b, MMR00, MR97b, Tou84]—they do not scale. The root cause for this limitation is their use of a quorum system [MR97a, Vuk10], i.e., sets of processes that are large enough to always intersect in at least one correct process. The size of a quorum scales linearly with the size of the system [CGR11]. To overcome this limitation, Malkhi *et al.* [MRWW01] generalized quorums to the probabilistic setting. In this setting, two random quorums intersect with a fixed, arbitrarily high probability, allowing the size of each quorum to be reduced to $O(\sqrt{N})$. Although we are not aware of any secure broadcast algorithm building on probabilistic quorums, such an algorithm could have a per-process communication complexity reduced from $O(N)$ to $O(\sqrt{N})$.

4.1.1 Samples

We present a probabilistic secure broadcast algorithm having $O(\log N)$ per-process communication and computation complexity, as well as broadcast latency. Essentially, we propose *samples* as a replacement for quorums. Like a probabilistic quorum, a sample is a randomly selected set of processes. Unlike quorums, however, samples do not need to intersect. Samples can be significantly smaller than quorums, as each sample must be large enough only to be *representative* of the system with high probability.

A process can use a sample to gather information about the global state of the system. To do so, we leverage the law of large numbers, trading performance for a fixed, arbitrarily small probability of the sample being non-representative (and the obtained information inaccurate). To get an intuition of the difference between quorums and samples, consider the emulation of a shared memory in message passing [ABND95]. One writes in a quorum

and reads from a quorum to fetch the last value written. Our algorithms are rather in the vein of "write all, read any". Here we would "write" by disseminating information using a gossip primitive and "read" by sampling the system to verify that a significant part of it has accepted the value.

We use samples to *estimate* the number of processes satisfying a set of *yes-or-no* properties, e.g., the number of processes that are ready to deliver a message m . To illustrate the idea, consider the case where a correct process π queries K randomly selected processes for a property P . Assume a fraction p of correct processes from the whole system satisfy property P . Let x be the fraction of positive responses (out of K) that π collects. By the Chernoff bound, the probability of $|x - p| \geq f + \epsilon$ is smaller or equal to $\exp(-\lambda(\epsilon)K)$, where λ quickly increases with ϵ . For a large enough sample K , the probability of x differing from p by more than $f + \epsilon$ can be made exponentially small.

To obtain samples, our algorithms use a sampling oracle, i.e., a black box returning the identity of a process from the system, picked with uniform probability. Implementing such an oracle is beyond the scope of this work, but it is straightforward in practice. In a permissioned system (i.e., one where the set of participating processes is known) sampling reduces to picking with uniform probability an element from the set of processes. In a permissionless system subject to Byzantine failures and limited churn, a (nearly) uniform sampling mechanism is available in literature (Bortnikov *et al.* [BGK⁺09]).

4.1.2 Scalable Secure Broadcast

We use samples as the core technique to design Ready Broadcast, a scalable secure broadcast algorithm. We generalize secure broadcast properties (validity, consistency, and totality, as mentioned earlier) to the probabilistic setting, allowing each property to be violated with a fixed, arbitrarily small probability ϵ . Ready Broadcast has logarithmic latency. For a fixed security parameter ϵ , the communication complexity grows logarithmically with N .

The Ready Broadcast itself is, at a high level, structured similarly to Bracha's deterministic algorithm [Bra87]—in 3 phases. In Bracha's broadcast, the first phase simply distributes the message, the second phase ensures consistency using a Byzantine quorum, and the third phase implements totality using a feedback loop and a quorum. Each phase involves $O(N)$ messages—in the first phase only for the sender, in the other phases for every correct process. We achieve lower per-process complexity $O(\log(n))$ (and thus scalability) by using gossip for the first phase and samples instead of quorums for the remaining phases.

In Bracha's broadcast, a process proceeds to the next phase when it received messages from a quorum of other processes. These messages provide a (deterministic) guarantee that the system reached a certain state, making it possible for the process to move on to the next protocol phase. In our case, a process does not require full assurance that the system reached the required state for passing to the next phase of the protocol. Instead, the process contents

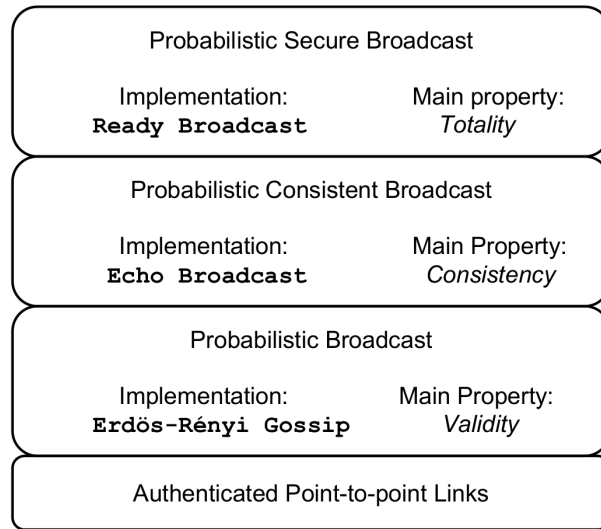


Figure 4.1 – Layered view of our broadcast abstractions. Starting from authenticated point-to-point links, we implement secure broadcast using 3 layers of broadcast abstractions. Intuitively, each layer is mainly responsible for guaranteeing one of secure broadcast’s properties

itself with a sufficiently high probability that this is the case and moves on.

We build probabilistic secure broadcast incrementally from weaker abstractions that we layer on top of each other as shown in Figure 4.1. Each layer corresponds to one phase of the broadcast algorithm. At the high level, the layers are as follows.

1. **Probabilistic broadcast** ensures simple dissemination of messages throughout the system.
2. **Probabilistic consistent broadcast** guarantees that, even in the case of a malicious sender, no two correct processes deliver two different messages (consistency).
3. **Probabilistic secure broadcast** adds the guarantee of totality—either all correct processes deliver a message or none does.

For each abstraction layer, we provide a corresponding implementation.

1. Erdős-Rényi Gossip is a probabilistic broadcast algorithm where each correct process relays the broadcast message to a randomly picked *gossip sample* of neighbors. This sample is larger than $\log(N)$, so the resulting gossip network is connected with high probability. Erdős-Rényi Gossip is used only for message dissemination. For a Byzantine sender, it provides no consistency guarantees.
2. Echo Broadcast is a probabilistic consistent broadcast algorithm in which the sender uses Erdős-Rényi Gossip to disseminate a message to every correct process. As Erdős-Rényi Gossip does not guarantee consistency, a correct process delivers the message

it received from Erdős-Rényi Gossip only upon receiving confirmation from a large enough fraction of its randomly selected *echo sample*. Echo Broadcast is used to deliver a consistent message to a subset of the correct processes. If the sender is Byzantine, however, probabilistic consistent broadcast provides no guarantee on the number of correct processes delivering the message.

3. Ready Broadcast is a probabilistic secure broadcast algorithm where the sender uses Echo Broadcast to disseminate a consistent message to a subset of the correct processes. Each correct process randomly picks a *ready sample* and a *delivery sample*. Upon receiving a message m from Echo Broadcast, a correct process becomes *ready* for m . If enough processes in the ready sample of a correct process π are ready for m , then π becomes ready for m as well, even if π itself has not delivered m from probabilistic consistent broadcast. This produces a *feedback* mechanism that, with high probability, converges to either only a few or all correct processes being ready for m . Finally, if enough processes in the delivery sample of a correct process π are ready for m , then π delivers m . With high probability, either all correct processes deliver m , or no correct process does. Ready Broadcast implements probabilistic secure broadcast in a Byzantine setting.

The rest of this chapter, after formally presenting the system model and assumptions, describes in detail the three abstractions just introduced (probabilistic broadcast, probabilistic consistent broadcast, and probabilistic secure broadcast) and their respective implementations (Erdős-Rényi Gossip, Echo Broadcast, and Ready Broadcast).

4.2 Model and Assumptions

In this section, we introduce the model underlying our protocols, and state the assumptions we make on the network as well as on the information available both to correct and Byzantine processes. We also introduce some useful notation.

We assume the following:

1. **(Processes)** The set Π of *processes* participating in the algorithm is **fixed**. Unless stated otherwise, we denote by $N = |\Pi|$ the total number of processes, and refer to the i -th process as $\pi_i \in \Pi$.
2. **(Failures)** At most a fraction f of the processes is *Byzantine*, i.e., subject to arbitrary failures. Byzantine processes are under the control of the same adversary, and can take coordinated action.

Unless stated otherwise, we denote by $\Pi_C \subseteq \Pi$ the set of correct processes and by $C = |\Pi_C| = (1 - f)N$ the number of correct processes.

3. (**Links**) Any two processes can communicate via **reliable, authenticated, point-to-point** links [CGR11].
4. (**Asynchrony**) Byzantine processes have control over the *network scheduling*, and can cause **arbitrary** but **finite delays** on any link, including links between pairs of correct processes.
5. (**Anonymity**) Byzantine processes cannot determine which correct processes a correct process is communicating with.
6. (**Randomness**) Every correct process has direct access to a local, unbiased, independent source of randomness. The Byzantine adversary does not have access to the output of the local source of randomness of any correct process.
7. (**Sampling**) Every correct process has direct access to an oracle Ω that, provided with an integer $n \leq N$, yields the identities of n distinct processes, chosen uniformly at random from Π using the local source of randomness.

Assumption 1 (Processes) is later weakened into an inequality, as we generalize our results to systems with churn. Assumption 4 (Asynchrony) represents one of the main strengths of this work: messages can be delayed arbitrarily and maliciously without compromising the safety properties of any of the algorithms presented in this work. Assumption 5 (Anonymity) represents the strongest constraint we put on the knowledge of the adversary. We later show that, without this assumption, an adversary could easily poison the view of the system of a targeted correct process without having to access the local randomness source of any correct process. Even against ISP-grade adversaries, Assumption 5 can be implemented in practice by means of, e.g., *onion routing* [DMS04] or *private messaging* [vdHLZZ15] algorithms. Assumption 7 (Sampling) reduces, in the permissioned case, to randomly sampling an list of processes available to each process. In a permissionless system subject to Byzantine failures and limited churn, a (nearly) uniform sampling mechanism is available in literature [BGK⁺09].

4.3 Probabilistic broadcast

In this section, we introduce the probabilistic broadcast abstraction and its properties. We then present Erdős-Rényi Gossip, an algorithm that implements probabilistic broadcast, and discuss its correctness. Here we only provide the high-level intuitions behind correctness arguments. The full formal analysis and correctness proof are included in separate work that is not part of this thesis [GKM⁺18].

The probabilistic broadcast abstraction serves the purpose of reliably broadcasting a single message from a designated correct sender to all correct processes. We use probabilistic broadcast in the implementation of Echo Broadcast (see Section 4.4) to initially distribute the message from the designated sender to all correct processes.

4.3.1 Definition

Let σ be the dedicated broadcasting process. The probabilistic broadcast interface exports the following events:

- **Request:** *Broadcast(m)*: Broadcasts a message m to all processes. This is only used by σ .
- **Indication:** *Deliver(m)*: Delivers a message m broadcast by process σ .

For any $\epsilon \in [0, 1]$, we say that probabilistic broadcast is ϵ -secure if:

1. **No duplication:** No correct process delivers more than one message.
2. **Integrity:** If a correct process delivers a message m , and σ is correct, then m was previously broadcast by σ .
3. **ϵ -Validity:** If σ is correct, and σ broadcasts a message m , then σ eventually delivers m with probability at least $(1 - \epsilon)$.
4. **ϵ -Totality:** If a correct process delivers a message, then every correct process eventually delivers a message with probability at least $(1 - \epsilon)$.

4.3.2 Algorithm

Erdős-Rényi Gossip (Algorithm 6) distributes a single message across the system by means of gossip: upon reception, a correct process relays the message to a set of randomly selected neighbors. The algorithm depends on one integer parameter, G —the *expected gossip sample size*), whose value we discuss in Section 4.3.3.

Initialization Upon initialization, (line 12) every correct process randomly samples a value \bar{G} from a *Poisson* distribution with expected value G , and uses the sampling oracle Ω to select \bar{G} distinct processes that it will use to initialize its *gossip sample* \mathcal{G} .

Link reciprocation Once its gossip sample is initialized, a correct process sends a `GossipSubscribe` message to all the processes in \mathcal{G} (line 14). Upon receiving a `GossipSubscribe` message from a process π (line 17), a correct process adds π to its own gossip sample (line 22), and sends back the gossiped message if it has already received it (line 20).

Gossip When broadcasting the message (line 25), a correct designated sender σ signs the message and sends it to every process in its gossip sample \mathcal{G} (line 36). Upon receiving a

Chapter 4. Probabilistic Secure Broadcast

Algorithm 6 Erdős-Rényi Gossip

```
1: Implements:
2:   Probabilistic broadcast
3:
4: Parameters:
5:    $G$ : expected gossip sample size
6:
7: Local variables:
8:    $\mathcal{G}$ : gossip sample ▷ Set of expected size  $G$ 
9:   delivered: delivered message, initially  $\perp$ 
10:
11: upon Init:
12:    $\mathcal{G} = \Omega(\text{Poisson}[G]);$ 
13:   for all  $\pi \in \mathcal{G}$  do
14:     Send (GossipSubscribe) to  $\pi$ 
15:   end for
16:
17: upon Receive (GossipSubscribe) from  $\pi$ :
18:   if delivered  $\neq \perp$  then
19:     (message, signature) = delivered;
20:     Send (Gossip, message, signature) to  $\pi$ 
21:   end if
22:    $\mathcal{G} \leftarrow \mathcal{G} \cup \{\pi\};$ 
23:
24: upon Broadcast(message): ▷ only process  $\sigma$ 
25:   dispatch(message, sign(message));
26:
27: upon Receive (Gossip, message, signature) from  $\pi$ :
28:   if verify( $\sigma$ , message, signature) then
29:     dispatch(message, signature);
30:   end if
31:
32: procedure DISPATCH(message, signature)
33:   if delivered =  $\perp$  then
34:     delivered  $\leftarrow$  (message, signature);
35:     for all  $\pi \in \mathcal{G}$  do
36:       Send (Gossip, message, signature) to  $\pi$ 
37:     end for
38:     trigger Deliver(message)
39:   end if
40: end procedure
```

correctly signed message from σ (line 28) for the first time (this is enforced by updating the value of *delivered*, line 33), a correct process delivers it (line 38) and forwards it to every process in its gossip sample (line 36).

4.3.3 Correctness

In this section, we provide the intuitions behind the correctness of Erdős-Rényi Gossip.

No duplication, integrity and validity

Erdős-Rényi Gossip always satisfies **no duplication, integrity and validity**:

- **No duplication:** A correct process maintains a *delivered* variable that it checks and updates before delivering a message. This prevents any correct process from delivering more than one message.
- **Integrity:** Before broadcasting a message, the sender signs that message with its private key. Before delivering a message m , a correct process verifies m 's signature. Under the assumption that signatures cannot be forged, this prevents any correct process from delivering a message that was not previously broadcast by the sender.
- **Validity:** Upon broadcasting a message, the sender also immediately delivers it. Since this happens *deterministically*, and thus Erdős-Rényi Gossip satisfies 0-validity, independently from the parameter G .

Totality

Erdős-Rényi Gossip satisfies ϵ_t -totality with ϵ_t upper-bounded by a function that decays exponentially with G , and polynomially increases with f .

Indeed, the network of connections established among the correct processes is an undirected Erdős-Rényi graph, and totality is satisfied if such graph is connected. This allows us to bound the probability of totality not being satisfied, using a well-known result on the connectivity of Erdős-Rényi graphs.

Sampling Upon initialization, a correct process randomly selects a sample of other processes with which it will exchange messages.

Link reciprocation We start by noting that every link is eventually reciprocated by correct processes, i.e., if a correct process π is in the sample of ρ , then ρ will eventually be in the sample of π (this is due to the fact that messages are always eventually delivered, see Assumption 4).

Correct connectedness We consider the sub-graph of connections only between *correct* processes. This sub-graph is eventually undirected. If this sub-graph is connected, then Erdős-Rényi Gossip satisfies totality. This is due to the fact that every message will eventually propagate through all the gossip links, reaching every correct process (again, due to Assumption 4).

Erdős-Rényi graph Any two correct processes have an independent probability of being connected. This is due to the fact that, upon initialization, the number of elements in a correct process' gossip sample is sampled from a Poisson distribution. Poisson distributions quickly limit to binomial distributions for large systems, and selecting a binomially distributed number of distinct objects from a set is equivalent to selecting each object with an independent probability. This proves that the sub-graph of connections between correct processes is an Erdős-Rényi graph.

Totality Erdős-Rényi graphs are known to display a connectivity *phase transition* [AO09]: when the expected number of connections each node has exceeds the logarithm of the number of nodes, the probability of the graph being connected steeply increases from 0 to 1 (in the limit of infinitely large systems, this increase becomes a step function). If we choose the algorithm parameter G sufficiently large (at least logarithmic in the number of processes), the probability of the sub-graph of correct processes being connected and, consequently, of Erdős-Rényi Gossip satisfying totality, is close to 1. Even for very large systems, already very small values for G provide negligible probabilities of violating the connectedness property. With increasing gossip sample size G , this probability diminishes exponentially.

4.4 Probabilistic consistent broadcast

In this section, we introduce the probabilistic consistent broadcast abstraction and discuss its properties. We then present Echo Broadcast, an algorithm that implements probabilistic consistent broadcast, and discuss its correctness. Again, the full formal analysis and correctness proof in [GKM⁺18].

The probabilistic consistent broadcast abstraction allows a subset of the correct processes to agree on a single message from a potentially Byzantine designated sender. Compared to probabilistic broadcast, probabilistic consistent broadcast sacrifices totality in favor of consistency. This means that, if the sender σ is malicious, it may happen that some, but not all correct processes deliver the broadcast message. Instead, probabilistic consistent broadcast guarantees that, even if the sender is Byzantine, no two correct processes deliver different messages. For a correct sender, probabilistic consistent broadcast behaves as probabilistic broadcast.

4.4.1 Definition

Let σ be the dedicated broadcasting process. The probabilistic consistent broadcast interface exports the following events:

- **Request:** $Broadcast(m)$: Broadcasts a message m to all processes. This is only used by σ .
- **Indication:** $Deliver(m)$: Delivers a message m broadcast by process σ .

For any $\epsilon \in [0, 1]$, we say that probabilistic consistent broadcast is ϵ -secure if:

1. **No duplication:** No correct process delivers more than one message.
2. **Integrity:** If σ is correct and a correct process delivers a message m , then m was previously broadcast by σ .
3. **ϵ -Total validity:** If σ is correct, and σ broadcasts a message m , every correct process eventually delivers m with probability at least $(1 - \epsilon)$.
4. **ϵ -Consistency:** Every correct process that delivers a message delivers the same message with probability at least $(1 - \epsilon)$.

4.4.2 Algorithm

Algorithm 7 Procedure *sample*

```

1: procedure SAMPLE(message, size)
2:    $\psi = \emptyset$ ; ▷  $\psi$  is a multiset.
3:   for size times do
4:      $\psi \leftarrow \psi \cup \Omega(1)$ ;
5:   end for
6:   for all  $\rho \in \psi$  do
7:     Send (message) to  $\rho$ 
8:   end for
9:   return  $\psi$ ;
10: end procedure

```

The intuition behind the Ready Broadcast algorithm is that before delivering a message m , a process p waits until a significant part of the system “promises” to not deliver a message different from m . To obtain this promise in a scalable way, p takes a uniformly random sample from all the processes in the system and only asks those to make this promise. A process makes such a promise by sending an Echo message.

Algorithm 7 defines a *SAMPLE* procedure that we use both in the implementation of Echo Broadcast and Ready Broadcast. Procedure *SAMPLE*(*message*, *size*) uses Ω to pick *size*

Chapter 4. Probabilistic Secure Broadcast

Algorithm 8 Echo Broadcast

```
1: Implements:
2:   Probabilistic consistent broadcast
3:
4: Parameters:
5:    $E$ : echo sample size
6:    $\hat{E}$ : delivery threshold
7:
8: Local variables:
9:    $\mathcal{E}$ : echo sample (a multiset), initially  $\emptyset$ 
10:   $\tilde{\mathcal{E}}$ : echo subscription set, initially  $\emptyset$ 
11:   $echo$ : echoed message, initially  $\perp$ 
12:   $replies$ : received echoes, initially  $\{\perp\}^E$ 
13:   $delivered$ : boolean, initially false
14:
15: Shared abstractions:
16:   $pb$ : instance of probabilistic broadcast
17:
18: upon Init:
19:    $\mathcal{E} = \text{SAMPLE}(\text{EchoSubscribe}, E)$ ;
20:
21: upon Receive EchoSubscribe from  $\pi$ :
22:   if  $echo \neq \perp$  then
23:      $(message, signature) = echo$ ;
24:     Send (Echo,  $message, signature$ ) to  $\pi$ 
25:   end if
26:    $\tilde{\mathcal{E}} \leftarrow \tilde{\mathcal{E}} \cup \{\pi\}$ ;
27:
28: upon Broadcast(message): ▷ only process  $\sigma$ 
29:   trigger  $pb.\text{Broadcast}(\text{Send}, message, sign(message))$ ;
30:
31: upon  $pb.\text{Deliver}(\text{Send}, message, signature)$ :
32:   if  $verify(\sigma, message, signature)$  then
33:      $echo \leftarrow (message, signature)$ ;
34:     for all  $\rho \in \tilde{\mathcal{E}}$  do
35:       Send (Echo,  $message, signature$ ) to  $\rho$ 
36:     end for
37:   end if
38:
39: upon Receive (Echo, message, signature) from  $\pi$ :
40:   if  $\pi \in \mathcal{E}$  and  $replies[\pi] = \perp$  and  $verify(\sigma, message, signature)$  then
41:      $replies[\pi] \leftarrow (message, signature)$ ;
42:   end if
43:
44: upon  $echo \neq \perp$  and  $|\{\rho \in \mathcal{E} \mid replies[\rho] = echo\}| \geq \hat{E}$  and  $delivered = false$  do
45:    $delivered \leftarrow true$ ;
46:    $(message, signature) = echo$ ;
47:   trigger Deliver( $message$ );
48:
```

processes with replacement, and sends them *message*. *SAMPLE* returns a multiset with selected processes.

Algorithm 8 describes Echo Broadcast, an implementation of probabilistic consistent broadcast. Echo Broadcast consistently distributes a single message across the system as follows:

- Initially, we use probabilistic broadcast to distribute potentially conflicting copies of the broadcast message to every correct process.
- Upon delivering a message m from probabilistic broadcast, a correct process issues an Echo message for m .
- Upon receiving enough Echoes for the message m it itself Echoed, a correct process delivers m .

A correct process collects Echo messages from a randomly selected *echo sample* of size E , and delivers the message it Echoed upon receiving \hat{E} Echoes for it. We discuss the values of the two parameters of Echo Broadcast in Section 4.4.3.

Sampling Upon initialization (line 18), a correct process randomly selects an *echo sample* \mathcal{E} of size E . Samples are selected with replacement by repeatedly calling Ω (Algorithm 7, line 4). A correct process sends an EchoSubscribe message to all the processes in its echo sample (Algorithm 7, line 7).

Publish-subscribe Unlike in Bracha's deterministic algorithm, where a correct process broadcasts its Echo messages to the whole system, here each process only listens for messages coming from its echo sample (line 40).

A correct process maintains an *echo subscription set* $\tilde{\mathcal{E}}$. Upon receiving an EchoSubscribe message from a process π , a correct process adds π to $\tilde{\mathcal{E}}$ (line 26). If a correct process receives an EchoSubscribe message *after* publishing its Echo message, it also sends back the previously published message (line 24).

A correct process only sends its Echo messages (line 35) to its echo subscription set.

Echo The designated sender σ initially broadcasts its message using probabilistic broadcast (line 29). Upon pb.Delivery of a message m (correctly signed by σ) (line 31), a correct process sends an Echo message for m to all the nodes in its echo subscription set (line 35). Note that by the properties of probabilistic broadcast, each process only delivers one message from probabilistic broadcast and thus only ever echoes a single message.

Delivery A correct process π that Echoed a message m delivers m (line 47) upon collecting at least \hat{E} Echo messages for m (line 44) from the processes in its echo sample. If a process has been sampled multiple times by π , its echo message accordingly counts multiple times.

4.4.3 Correctness

In this section, we provide the intuitions behind the correctness of Echo Broadcast. For the probabilistic properties of probabilistic consistent broadcast, correctness of Echo Broadcast reduces to showing that the probability of failure can be bounded. The bound ϵ then determines the probabilistic consistent broadcast's ϵ -security.

No duplication and integrity

Echo Broadcast always satisfies **no duplication** and **integrity**:

- **No duplication:** A correct process maintains a *delivered* variable that it checks and updates before delivering a message. This prevents any correct process from delivering more than one message.
- **Integrity:** Before broadcasting a message, the sender signs that message with its private key. Before delivering a message m , a correct process verifies m 's signature. Under the assumption that signatures cannot be forged, this prevents any correct process from delivering a message that was not previously broadcast by the sender.

Total validity

We start by noting that in our algorithm a correct process will only deliver a message it Echoed. Thus, if the totality of probabilistic broadcast is compromised, then the total validity of probabilistic consistent broadcast is compromised as well. However, by the validity and totality property of probabilistic broadcast, every correct process eventually pb.Delivers the only message m broadcast by the correct sender σ (σ is correct by the premise of total validity).

By the algorithm, every correct process eventually sends Echo messages for m to its echo subscription set $\hat{\mathcal{E}}$. Therefore, a correct process will deliver m if no more than $E - \hat{E}$ elements of its sample are Byzantine. The probability of a correct process delivering m is thus equal to the probability of that process randomly picking at least \hat{E} correct processes for its echo sample. Under the assumption that samples are selected independently (Assumption 6), we can compute a lower bound on the probability of every correct process delivering m , i.e., Echo Broadcast satisfying total validity.

Consistency

We only provide a high level intuition of the analysis of probabilistic consistent broadcast. The full analysis is part of separate work [GKM⁺18].

By the definition of consistency, the goal of the adversary is to have at least two correct processes deliver a different message each. Under the assumptions stated in Section 4.2, one can make (formally provable) claims about the optimal strategy of the Byzantine adversary. In particular, since the correct processes' echo samples are uniformly random and secret (Assumption 6) and the adversary cannot observe which correct processes communicate (Assumption 5), it is indistinguishable for the adversary which correct process is which. Therefore, when the adversary chooses its next action, it has equal probability of success regardless of which correct process that action targets.

It can also be shown that the optimal adversarial strategy first makes at least one correct process deliver message m by making randomly picked correct processes pb.Deliver m until some of them delivers m . Then, it makes all other correct processes pb.Deliver $m' \neq m$, hoping that this will be sufficient for some of them to deliver m' . Provably, no other strategy achieves a higher probability of the adversary's success.

Having constrained the optimal adversarial strategy this way, it is possible to derive an upper bound on the probability of compromising the consistency of probabilistic consistent broadcast, as a function of the system size, the fraction of processes assumed to be Byzantine, and the algorithm parameters (gossip sample size, echo sample size, etc.). Moreover, and particularly importantly, in order to obtain a constantly low probability of compromising consistency, the sample sizes (and thus the computation and communication overhead of a process) only need to be logarithmic in the system size.

4.5 Probabilistic secure broadcast

In this section, we finally introduce the probabilistic secure broadcast abstraction and describe its properties. We then present `Ready Broadcast`, an algorithm that implements probabilistic secure broadcast, and discuss its correctness.

The probabilistic secure broadcast abstraction allows the entire set of correct processes to agree on a single message from a potentially Byzantine designated sender. Probabilistic secure broadcast is a strictly stronger abstraction than probabilistic consistent broadcast. While in the case of a Byzantine sender, probabilistic consistent broadcast only guarantees that every correct process that delivers a message delivers the same message (consistency), probabilistic secure broadcast also guarantees that either all correct processes deliver this message or none of them does (totality).

4.5.1 Definition

Let σ be the dedicated broadcasting process. The probabilistic secure broadcast interface exports the following events:

- **Request:** $Broadcast(m)$: Broadcasts a message m to all processes. This is only used by σ .
- **Indication:** $Deliver(m)$: Delivers a message m broadcast by process σ .

For any $\epsilon \in [0, 1]$, we say that probabilistic secure broadcast is ϵ -secure if:

1. **No duplication:** No correct process delivers more than one message.
2. **Integrity:** If a correct process delivers a message m , and σ is correct, then m was previously broadcast by σ .
3. ϵ -**Validity:** If σ is correct and σ broadcasts a message m , then σ eventually delivers m with probability at least $(1 - \epsilon)$.
4. ϵ -**Totality:** If a correct process delivers a message, then every correct process eventually delivers a message with probability at least $(1 - \epsilon)$.
5. ϵ -**Consistency:** Every correct process that delivers a message delivers the same message with probability at least $(1 - \epsilon)$.

4.5.2 Algorithm

We design `Ready Broadcast` as another layer on top of probabilistic consistent broadcast. The main challenge is to guarantee totality on top of the properties of probabilistic consistent broadcast. The principal idea is that, similarly to probabilistic broadcast, a correct process does not deliver a message m received through the underlying layer immediately, but instead waits for some form of (probabilistic) confirmation. In probabilistic consistent broadcast, this confirmation says that (with high probability) no other correct process will ever deliver a message different from m . In probabilistic secure broadcast, a correct process, before delivering m , needs to wait for the confirmation that (with high probability) all other correct processes will eventually deliver m .

In order to obtain this confirmation, we use the notion of a process being *ready* to deliver m . We employ a mechanism similar to the spreading of a contagious disease that makes sure that if a critical fraction of the system is ready for m , all correct processes will eventually be ready for m . A process delivers m when it discovers that this critical fraction has been reached. To discover in a scalable way whether this critical fraction has been reached, every correct process takes a representative sample of the whole system and evaluates whether the critical

Algorithm 9 Ready Broadcast

```

1: Implements:
2:   ProbabilisticSecureBroadcast, instance psb
3:
4: Parameters:
5:    $R$ : ready sample size            $\hat{R}$ : contagion threshold
6:    $D$ : delivery sample size        $\hat{D}$ : delivery threshold
7:
8: Local variables:
9:    $\tilde{\mathcal{R}}$ : ready subscription set, initially  $\emptyset$ 
10:   $\mathcal{R}$ : ready sample (a multiset), initially  $\emptyset$ 
11:   $\mathcal{D}$ : delivery sample (a multiset), initially  $\emptyset$ 
12:  ready: ready set, initially  $\emptyset$ 
13:  replies.ready: ready messages from ready sample, initially  $\{\emptyset\}^R$ 
14:  replies.delivery: ready messages from delivery sample, initially  $\{\emptyset\}^D$ 
15:  delivered: boolean, initially false
16:
17: Shared abstractions:
18:  pcb: instance of probabilistic consistent broadcast
19:
20: upon Init:
21:    $\mathcal{R} = \text{SAMPLE}(\text{ReadySubscribe}, R)$ ;
22:    $\mathcal{D} = \text{SAMPLE}(\text{ReadySubscribe}, D)$ ;
23:
24: upon Receive (ReadySubscribe) from  $\pi$ :
25:   for all (message, signature)  $\in$  ready do
26:     Send (Ready, message, signature) to  $\pi$ 
27:   end for
28:    $\tilde{\mathcal{R}} \leftarrow \tilde{\mathcal{R}} \cup \{\pi\}$ ;
29:
30: upon Broadcast(message) ▷ only process  $\sigma$ 
31:   trigger pcb.Broadcast(Send, message, sign(message));
32:
33: upon pcb.Deliver (Send, message, signature):
34:   if verify( $\sigma$ , message, signature) then
35:     ready  $\leftarrow$  ready  $\cup$  {(message, signature)};
36:     for all  $\rho \in \tilde{\mathcal{R}}$  do
37:       Send (Ready, message, signature) to  $\rho$ 
38:     end for
39:   end if
40:

```

Chapter 4. Probabilistic Secure Broadcast

```
41: upon Receive (Ready, message, signature) from  $\pi$ :
42:   if verify( $\sigma$ , message, signature) then
43:     reply = (message, signature);
44:     if  $\pi \in \mathcal{R}$  then
45:       replies.ready[ $\pi$ ]  $\leftarrow$  replies.ready[ $\pi$ ]  $\cup$  {reply};
46:     end if
47:     if  $\pi \in \mathcal{D}$  then
48:       replies.delivery[ $\pi$ ]  $\leftarrow$  replies.delivery[ $\pi$ ]  $\cup$  {reply}
49:     end if
50:   end if
51:
52: upon exists message such that
    $|\{\rho \in \mathcal{R} \mid (\text{message}, \text{signature}) \in \text{replies.ready}[\rho]\}| \geq \hat{R}$  do
53:   ready  $\leftarrow$  ready  $\cup$  {(message, signature)};
54:   for all  $\rho \in \tilde{\mathcal{R}}$  do
55:     Send (Ready, message, signature) to  $\rho$ 
56:   end for
57:
58: upon exists message such that
    $|\{\rho \in \mathcal{D} \mid (\text{message}, \text{signature}) \in \text{replies.delivery}[\rho]\}| \geq \hat{D}$  and delivered = false do
59:   delivered  $\leftarrow$  true;
60:   (message, signature) = echo;
61:   trigger Deliver(message)
62:
```

fraction of the sample (instead of the whole system) is ready to deliver m . When this occurs, the process can itself deliver m .

Algorithm 9 lists the complete implementation of Ready Broadcast. For a correct process π and a message m broadcast by σ , Ready Broadcast securely distributes m across the system as follows:

- Initially, we use probabilistic consistent broadcast to consistently distribute m to a subset of the correct processes. Note that if σ is correct, then by total validity of probabilistic consistent broadcast all correct processes receive m . However, for a faulty σ , this subset of correct processes might be a proper subset of Π .
- A processes π becomes ready for m and consequently issues a Ready message for m when:
 - π receives m directly through probabilistic consistent broadcast, or
 - π collects enough (\hat{R}) Ready messages for m from its *ready sample*.
- π delivers m if m is the first message for which π collected enough (\hat{D}) Ready messages from its delivery sample.

A correct process collects Ready messages from two randomly selected samples, the *ready sample* of size R , and the *delivery sample* of size D . It issues a Ready message for m upon collecting \hat{R} Ready messages for m from its ready sample, and it delivers m upon collecting \hat{D} Ready messages for m from its delivery sample. The values R, \hat{R}, D, \hat{D} are parameters of the algorithm that determine the implementation's ϵ -security. In practice, R and D , the respective ready and delivery sample sizes, determine how many other processes a correct process has to communicate with. It can be shown (see [GKM⁺18]) that, for any required ϵ -security, these parameters only need to grow logarithmically with the size of the system.

The execution of the protocol proceeds in the following phases:

Sampling Upon initialization (line 20), a correct process randomly selects a ready sample \mathcal{R} of size R , and a delivery sample \mathcal{D} of size D . Samples are selected with replacement by repeatedly calling Ω (Algorithm 7, line 4).

Publish-subscribe Like Echo Broadcast, Ready Broadcast uses publish-subscribe to reduce its communication complexity. This is achieved by having each correct process send Ready messages only to its ready subscription set (lines 36 and 54), and accept Ready messages only from its ready and delivery samples (lines 44 and 47).

Consistent broadcast The designated sender σ initially broadcasts its message using probabilistic consistent broadcast (line 31). Upon `pcb.Delivery` of a message m (correctly signed by σ) (line 33), a correct process sends a Ready message for m (line 37) to all the processes in its ready subscription set.

Contagion Upon collecting \hat{R} Ready messages for a message m (line 52), a correct process sends a Ready message for m (line 55) to all the nodes in its ready subscription set. This behavior, mimics the spreading of a contagious disease in a population. A process that becomes ready for m (“infected”), becomes “contagious” itself and may potentially “infect” other processes, i.e., make them ready for m as well.

Delivery Upon collecting \hat{D} Ready messages for a message m for the first time, (line 58), a correct process delivers m (line 61).

4.5.3 Correctness

In this section, we provide an intuitive correctness argument for Ready Broadcast. For the probabilistic properties of probabilistic secure broadcast, correctness of Ready Broadcast reduces to showing that the probability of failure can be bounded. The bound ϵ then determines the probabilistic secure broadcast’s ϵ -security.

No duplication and integrity

Ready Broadcast always satisfies no duplication and integrity:

- **No duplication:** A correct process maintains a *delivered* variable that it checks and updates before delivering a message. This prevents any correct process from delivering more than one message.
- **Integrity:** Before broadcasting a message, the sender signs that message with its private key. Before delivering a message m , a correct process verifies m ’s signature. Under the assumption that signatures cannot be forged, this prevents any correct process from delivering a message that was not previously broadcast by the sender.

Validity

We obtain an upper bound on the probability of compromising validity by assuming that, if the total validity of `pcb` is compromised, then the validity of `psb` is compromised as well. This probability comes directly by the ϵ -security of probabilistic consistent broadcast. When `pcb` satisfies total validity and the sender σ is correct (assumed by the premise of the validity property), then every correct process issues a Ready message for the same message m .

If σ has at least \hat{D} correct processes in its delivery sample, it delivers m . Therefore, the probability of compromising validity is bound by the probability of σ having more than $D - \hat{D}$ Byzantine processes in its delivery sample. We compute the probability of this happening by noting that, since each delivery sample is independently picked with replacement, the number of Byzantine processes in any delivery sample is independently, binomially distributed.

Consistency

The probability of compromising consistency can be upper-bounded by assuming that, if the consistency of pcb is compromised, then the consistency of psb is compromised as well.

When pcb does satisfy consistency, at most one message m^* is pcb.Delivered by any correct process. It is thus easy for the adversary to make some correct process deliver m^* . To achieve this, it is sufficient that some malicious nodes follow the protocol. We thus assume from now on that the adversary can always make any correct node deliver m^* .

Therefore, consistency is compromised if and only if at least one correct process delivers a message $m \neq m^*$, given that no correct process pcb.Delivers m (we discuss the case where pcb does satisfy consistency).

We start by noting that, since a correct process can be ready for an arbitrary number of messages, the propagation of “readiness” for a message m is not affected by the propagation of another message m' . More precisely, let π be a correct process. If enough processes in π 's delivery sample are eventually ready both for m and m' , then π can deliver m or m' . Whether π delivers m or m' is determined only by the network scheduling of the system (which can be arbitrary, see Assumption 4).

The probability of m being delivered by any correct process, given that no correct process pcb.Delivers m , is maximized when every Byzantine process issues a Ready message for m . Note how a Byzantine process issuing a Ready message for m behaves identically to a correct process that pcb.Delivered m . One can use a contagion model to compute the probability of any correct process delivering m . We model the system as a population where π being in ρ 's ready sample means that π can potentially infect ρ . All Byzantine processes are initially infected by (i.e., ready for) m .

Given the total number of (correct and malicious) processes that are ready for m at the end of such a contagion process, one can compute the probability that at least one correct process can deliver m . Finally, we note that:

- If a correct process can deliver m , then it can deliver any other message $m' \neq m^*$ as well. Since m^* can already be delivered by every correct process, however, this does not affect the probability of consistency being compromised.
- If a correct process cannot deliver m , then it cannot deliver any other message $m' \neq m^*$.

This is due to the fact that the ready and delivery samples of each correct process are unchanged when processing m and m' .

Totality

We compute an upper bound on the probability of compromising totality by assuming that, if the consistency of pcb is compromised, then the totality of psb is compromised as well. When pcb satisfies consistency, at most one message m^* is pcb.Delivered by any correct process.

We further bound the probability of compromising totality by assuming that the Byzantine adversary can arbitrarily cause any correct process to pcb.Deliver m^* . In our algorithm, whenever a correct process π becomes ready for m^* as a result of having pcb.Delivered m^* , zero or more additional correct processes will also become ready for m^* as a result of having collected enough Ready messages for m^* . This happens either “directly” for processes which picked π in their ready sample, or “indirectly” by other processes infected from π issuing their own ready messages.

By definition, totality is not compromised by m^* if:

- no correct process delivers m^* or
- all correct processes deliver m^* .

Consequently, whenever the adversary makes a correct process pcb.Deliver m^* (potentially causing other correct processes to become ready for m^* as well):

- If no correct process delivered m^* , the probability of compromising totality is non-null only if the Byzantine adversary causes at least one more correct process to pcb.Deliver m^* .
- If all correct processes delivered m^* , then totality is not compromised by m^* .

The idea behind guaranteeing totality is to use the positive feedback effect in the contagion process. There is a critical threshold for the number of correct processes, such that:

- If fewer correct processes are ready for m^* , it is very unlikely that many other correct processes become ready through infection.
- If more correct processes are ready for m^* , it is very likely that all other processes become ready through infection.

If this threshold is low enough to make it unlikely for any correct process to deliver m^* , totality is unlikely to be compromised. If the adversary makes too few processes pcb.Deliver m^* (and

thus be ready for m^*), no correct process delivers m^* . Once the adversary makes enough correct processes ready for m^* to have a non-negligible chance of m^* being delivered by at least one correct process, with high probability it will have already triggered an avalanche of Ready messages making all correct processes deliver m^* .

Precise models and formal proofs of the intuitions stated above, as well as the quantifications of important probability values and their relation to the algorithm parameters are the subject of separate work that is not part of this thesis [GKM⁺18].

5 Atum: Scalable Group Communication Using Volatile Groups

This chapter presents Atum, a Byzantine fault tolerant group communication middleware for a large, dynamic, and hostile environment. At the heart of Atum lies the novel concept of *volatile groups*: small, dynamic groups of nodes, each executing a state machine replication protocol, organized in a flexible overlay. Using volatile groups, Atum scatters faulty nodes evenly among groups, and then masks each individual fault inside its group. To broadcast messages among volatile groups, Atum runs a gossip protocol across the overlay.

Atum can serve as a highly scalable communication layer for an asset transfer implementation that is also designed to support high node churn. Additionally, in accordance with our result presented in Section 2.4, since the members of each volatile group can solve consensus, Atum can be used to implement more powerful abstractions as well. The applicability of Atum is, however, more general than just a building block of an asset transfer system. In this chapter, we discuss it as a system in its own right, demonstrating its applicability using various applications.

We report on our synchronous and asynchronous (eventually synchronous) implementations of Atum, as well as on three representative applications that we build on top of it: A publish/subscribe platform, a file sharing service, and a data streaming system. We show that (a) Atum can grow at an exponential rate beyond 1000 nodes and disseminate messages in polylogarithmic time (conveying good scalability); (b) it smoothly copes with 18% of nodes churning every minute; and (c) it is impervious to arbitrary faults, suffering no performance decay despite 5.8% Byzantine nodes in a system of 850 nodes.

5.1 Introduction

Group communication services (GCSs) are a central theme in systems research [Bir85, CDK⁺03, CKV01, KT91, LCM⁺08]. These services provide the abstraction of a node *group*, and typically export operations for *joining* or *leaving* the group, as well as *broadcasting* messages inside this group. For a node in the group, the GCS acts as a middleware between the application and

the underlying communication stack. The application simply sends and receives messages, while the network topology, low-level communication protocols and the OS networking stack are abstracted away by the GCS.

A wide range of applications can be built using this abstraction, spanning from infrastructure services in datacenters [ACMP11, GvR13], to streaming and publish/subscribe engines in cooperative networks [CDK⁺03, CDKR02, KRAV03, LCM⁺08], or intrusion-tolerant overlays [CCC⁺05, JAVR06].

To cope with the needs of modern applications, GCSs need to be *scalable*, *robust*, and *flexible*. Scalability is a primary concern because many applications today involve thousands of nodes [OGP03] and serve millions of users [ACMP11, VGLN07]. The main indicator of scalability in a GCS is the cost of its operations, which should ideally be sublinear in system size.

Given the scale of these systems, faults inevitably occur on a daily basis. Crashed servers and buggy software with potentially arbitrary behavior are common in practice, both in cooperative systems (e.g., peer-to-peer) and datacenter services [amab, Dea09, NDO11, OGP03]. For instance, according to Barroso et al. [BCH13], in a 2000-node system, 10 nodes fail each day. Clearly, GCSs have to be robust by design.

GCSs also need to be flexible, tolerating a considerable fraction of nodes that join and leave the system, i.e., *churn*. Peer-to-peer services are naturally flexible [VGLN07]. For datacenter services, churn emerges as a consequence of power saving techniques, software updates, service migration, or failures [BCH13]; cross-datacenter deployment of services tends to further exacerbate the churn issue.

There are well-known techniques to address individually each of robustness, scalability, and flexibility. To obtain a robust design, the classic approach is *state machine replication* [Lam78, Sch90]: Several replicas of the service work in parallel, agreeing on operations they need to perform, using some consensus protocol [CL02]. State machine replication (SMR) is powerful enough to cope even with arbitrary, i.e., *Byzantine* faults [AMQ13, CL02, KAD⁺07].

To provide a scalable design, *clustering* is the common approach. The nodes of the system are partitioned in multiple groups, each group working independently; the system can grow by simply adding more groups [BPR14, GBKA11]. To achieve flexibility and handle churn, join and leave operations should be lightweight and entail small, localized changes to the system [LCM⁺08].

A standard approach to attain flexibility in a GCS is through *gossip protocols* [DGH⁺88]. With gossip, each participant periodically exchanges messages with a small, randomized subset of nodes. Gossip-based schemes disseminate messages efficiently, in logarithmic time and with logarithmic cost [DGH⁺88].

It is appealing to combine clustering with SMR and gossip to tackle all of the issues above. At

first glance, it seems natural to organize a very large system as a set of reliable groups and have them communicate through gossip. Unfortunately, this combination poses a major challenge due to a conflict between robustness and flexibility. Churn induces changes to the system structure and calls for groups that are highly dynamic in nature, i.e., fluctuate in number and size. In contrast, (Byzantine-resilient) SMR has opposing requirements, imposing strict constraints on every group, to keep groups robust and efficient. In particular, to ensure robustness, SMR requires a bounded fraction of faults in every group [BT85b, DS83]; to achieve efficiency, it is important to keep groups small in size, as SMR scales poorly due to quadratic communication complexity [CML⁺06].

In this chapter, we report on our experiences from designing and building *Atum*, a novel group communication middleware that seeks to overcome this challenge. To mitigate the above conflict, we introduce the notion of *volatile groups* (vgroups). These are clusters of nodes that are small (logarithmic in system size), dynamic (changing their composition frequently due to churn), yet robust (providing the abstraction of a highly available entity). Every vgroup executes a SMR protocol, confining each faulty node to that vgroup. Among vgroups, we use two additional protocols: *random walk shuffling* and *logarithmic grouping*.

Random walk shuffling ensures that faulty nodes, if any, are dispersed evenly among vgroups. Whenever a vgroup changes, e.g., due to nodes joining or leaving, *Atum* uses random walks to refresh the composition of this vgroup to contain a fresh, uniform sample of nodes from the whole system. This technique is particularly important for cases when faults accumulate over time in the same vgroup.¹ We thus refresh a vgroup after any node joins or leaves it.

Our *logarithmic grouping* protocol guarantees in addition that every vgroup has a size that is logarithmic in the system size. Whenever a vgroup becomes too large or too small, *Atum* splits or merges groups, to keep their size logarithmic.

To efficiently disseminate messages *among* vgroups, we use a gossip protocol. The complexity of gossip is known to be logarithmic in system size [DGH⁺88]. In *Atum*, we address each gossip to a vgroup of logarithmic size, resulting in an overall polylogarithmic complexity.

We implement two versions of *Atum*, one using a synchronous SMR algorithm, and one based on an asynchronous algorithm.² To illustrate the capabilities of *Atum*, we build three applications: *ASub*, a publish/subscribe service, *AShare*, a file sharing platform, and *AStream*, a data streaming system. Given these two implementations and the three applications, we report on their deployment over a variety of configurations in a single datacenter, as well as across multiple datacenters around the globe. We show that *Atum*: (a) supports an exponential growth rate, scaling well beyond 1000 nodes, and smoothly copes with 18% of nodes churning every minute; (b) is robust against arbitrary behavior, coping with 5.8% Byzantine nodes in

¹This may happen due to bugs [NDO11, OGP03, WECK07] or join-leave attacks [AS07]; both of these situations reflect the concentration of faults in the same group.

²Strictly speaking, an asynchronous SMR implementation is impossible [FLP85]. In this context, we call a SMR implementation asynchronous if it requires synchrony only for liveness (eventual synchrony), like PBFT [CL02].

a 850-node system; (c) disseminates messages in polylogarithmic time, incurring a small overhead compared to classical gossip.

It is important to note that the goal of our work is to explore the feasibility of a general-purpose GCS – in the same vein as Isis [Bir85], Amoeba [KT91], Transis [DM96], or Horus [VRBM96], but for large, dynamic, and hostile networks. Our experiences with Atum highlight the numerous complications that arise in a GCS designed for such an environment (Section 5.7). The protocols underlying the vgroup abstraction – SMR; group resizing, splitting and merging; distributed random walks – are challenging by themselves. Combining them engenders further complexity and trade-offs. We believe, however, that the vgroup abstraction is an appealing way to go, and we hope our experiences pave the way for new classes of GCSs, each specialized for their own needs. Note also that we do not argue that the vgroup abstraction (and its underlying protocols) is a silver bullet, necessary and sufficient for every part of an application that requires multicasting in a challenging environment. For instance, in our streaming application, we use Atum to reliably deliver small authentication metadata; to disseminate the actual stream data at high throughput, we use a separate multicast protocol.

To summarize, the main contributions presented in this chapter are the following:

- We introduce the notion of vgroups – small, dynamic, and robust clusters of nodes. The companion random walk shuffling and logarithmic grouping techniques ensure that every vgroup executes SMR efficiently, despite arbitrary faults and churn.
- Using a gossip protocol among vgroups, we design Atum, a GCS for large, dynamic, and hostile environments.
- We report on the implementation of two versions of Atum and three applications on top of it.

The remainder of this chapter is structured as follows. In Section 5.2, we describe the assumptions and guarantees of Atum. Section 5.3 and Section 5.4 present Atum’s design and the three applications we built on top of it, respectively. We move on to discussing some practical aspects of our synchronous and asynchronous Atum implementations in Section 5.5. Section 5.6 reports on our extensive experimental evaluation, using the above-mentioned three applications. We discuss our experiences and lessons learned in Section 5.7. Finally, we conclude in Section 5.8.

5.2 Assumptions and Guarantees

Atum addresses the problem of group communication in a large network. Despite arbitrary faults or churn, Atum guarantees the following properties for correct nodes.

- *Liveness*: If a node requests to join the system, then this node eventually starts to deliver

the messages being broadcast in the system; this captures the liveness of both join and broadcast operations.

- *Safety*: If some node delivers a message m from node v , then v previously broadcast m .

Atum uses SMR as a building block (inside every vgroup) and it inherits all assumptions made by the underlying SMR protocol. This also applies to assumptions related to reconfiguring the replicated state machine. We assume that a bounded number of nodes are subject to arbitrary failures such as bugs or crashes, so we consider SMR protocols with Byzantine fault tolerance guarantees. We also assume that these failures are uncorrelated. Depending on the target environment, we can use either an asynchronous protocol [CL02], or a synchronous one [DS83]. Atum itself, however, has a general design and is oblivious to the specifics of this protocol. As we will discuss later, we experiment with both versions of this protocol.

We model the system as a large, decentralized network, where a significant fraction of nodes can join and leave (i.e., churn). For liveness, we only expect the network to eventually deliver messages; this is a valid assumption even in highly unstable networks such as the Internet. Safety relies on the correctness of the underlying SMR protocol.

We use cryptography (public-key signatures and MACs) to authenticate messages, and assume that the adversary is computationally bounded and cannot subvert these techniques. We do not consider Sybil attacks in our model; these can be handled using well-known techniques, such as admission control [Dou02] or social connections [LLK10]. An alternative, decentralized solution to deter Sybil attacks is to rely on cryptographic proofs of work, as in Bitcoin [Nak08]. To initialize the process of joining the system, a node also requires one other node that it trusts and that is already member of the system.

Atum tolerates a limited number of nodes isolated by a network partition, in both the synchronous and asynchronous case. In practice, we treat isolated nodes as faulty, so the bound on the number of faults also includes partitioned nodes. Aside from the SMR algorithm, a severe network outage might break liveness, but not safety.

5.3 Design

Atum is a group communication middleware positioned between a distributed application and the underlying network stack. It has a layered design comprising four layers, as depicted in Figure 5.1. At the bottom, the *node* layer handles inter-node communication. We use standard techniques here – cryptographic algorithms to secure communication, and a network transport protocol for reliable inter-node message transmission. Since these are orthogonal to our design, we do not dwell on their details.

At the *group* layer (Section 5.3.1), Atum partitions nodes into vgroups of logarithmic size. We ensure the robustness of each vgroup using a state machine replication protocol with BFT

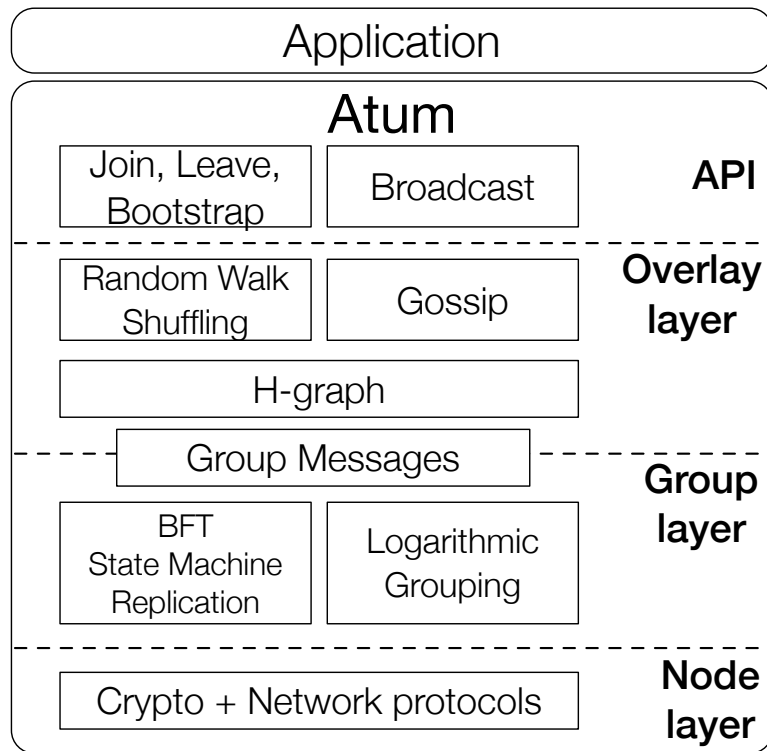


Figure 5.1 – Atum’s layered architecture.

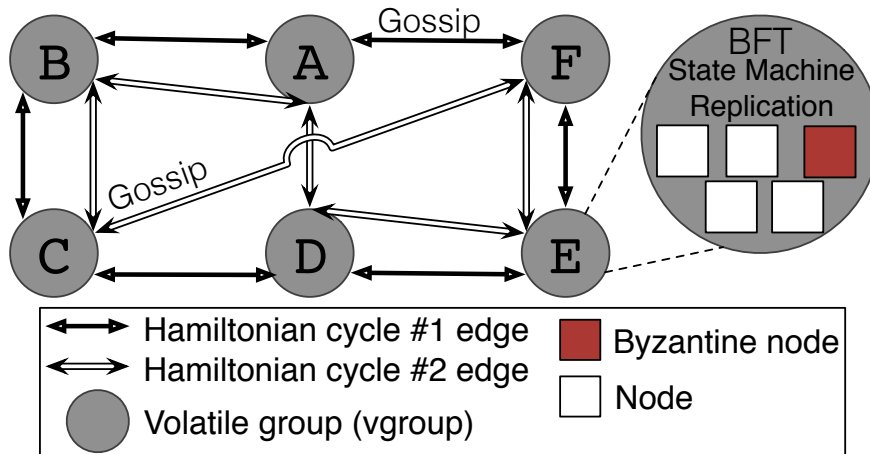


Figure 5.2 – An instance of Atum: Vgroups interconnected by an H-graph overlay with two cycles.

guarantees. For inter-vgroup communication we use special messages, called *group messages*, that ensure reliable communication for pairs of vgroups.

The *overlay* layer (Section 5.3.2) connects vgroups and enables them to communicate. The network formed by vgroups has the structure of an H-graph [LS03], as Figure 5.2 depicts. At this layer, the protocols are typically randomized (based on gossip and random walks), and rely on group messages.

At the topmost layer sits the Atum *API*. For membership management, we provide *bootstrap*, *join*, and *leave* operations; for data dissemination, we expose a *broadcast* operation. In the remaining parts of this section we describe the interplay between these layers and their corresponding techniques, including the API operations.

5.3.1 Group layer

The purpose of the group layer is to mask failures of individual nodes and provide the abstraction of robust vgroups. We partition nodes in vgroups of size g , and apply a BFT SMR protocol in every vgroup. We design Atum to be agnostic to this underlying protocol, so our system can support either a synchronous version, which tolerates at most $f = \lfloor (g - 1)/2 \rfloor$ faults in every vgroup [DS83], or an asynchronous version, which has a lower fault-tolerance at $f = \lfloor (g - 1)/3 \rfloor$ faults per vgroup [CL02]. We say that a vgroup is robust if the number of faults in that vgroup does not exceed f (for any configuration of the vgroup).

Assuming each node in a vgroup has the same constant probability of being faulty (we will discuss this assumption closely in Section 5.3.2), the size g of a vgroup is critical for ensuring its robustness. The more nodes a vgroup contains, the higher the probability of being robust. To get the intuition, consider a synchronous system with 1 failure out of every 20 nodes, i.e., with failure probability of 0.05. A vgroup with $g = 4$ nodes tolerates $f = \lfloor (4 - 1)/2 \rfloor = 1$ faults and fails with probability $Pr[X \geq 2] = 0.014$; the random variable X denotes the number of failures and follows the binomial distribution $X \sim B(4, 0.05)$. But a 20-node vgroup, with $f = 9$, will fail with $Pr[X \geq 10] = 1.134 \cdot 10^{-8}$. Thus, larger vgroups are more desirable from a robustness perspective. On the other hand, large vgroups entail a bigger overhead of the BFT protocol, penalizing performance [CML⁺06]. Efficiency thus requires a smaller g .

At the group layer, there is thus a clear trade-off between robustness (larger g) and performance (smaller g). Whatever vgroup size we pick, the probability of *all* vgroups being robust decreases as the number of vgroups in the system grows. To understand the trade-off, let us denote the expected system size by n , and the number of vgroups by n/g .³ Consider a growing system. At one extreme, if vgroup size g is constant, we promote efficiency at the cost of robustness; as the system grows and accumulates vgroups, the probability of *all* vgroups being robust diminishes. At the other extreme, if the number of vgroups n/g is constant, we favor robustness to the

³The expected system size n need not be exact, an estimation suffices. If n is conservative (too large), then the system trades efficiency for better robustness; and vice versa if n is too small.

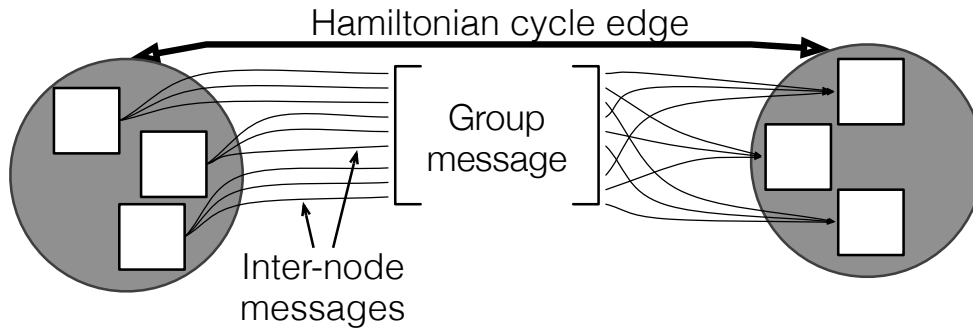


Figure 5.3 – Two vgroups communicate (e.g., gossiping) through a group message, which consists of multiple inter-node messages.

detriment of efficiency: Robustness improves as n grows, but the BFT overhead may become impractical. We argue that a middle-ground between these two extremes is the best option.

Logarithmic grouping. We favor both efficiency and robustness in a controlled manner by making g and n/g grow slowly, sublinearly in system size. We do so by setting $g = k \cdot \log(N)$, i.e., vgroups have their size logarithmic in the system size; it has previously been shown that this is a good efficiency/robustness trade-off [AS09, GHK13, Sch05]. The system parameter k controls the above-mentioned trade-off. With bigger k , robustness increases at the cost of performance, independently of system size. In practice, we believe $k = 4$ is a good trade-off: e.g., in a system with 6% *simultaneous* arbitrary faults, there is a probability of 0.999 of *all* vgroups being robust.

In a dynamic environment, vgroups do not have a fixed size g , but their size fluctuates due to churn. We introduce two system parameters, g_{\min} and g_{\max} , defined by a system administrator at startup; these define the minimum and maximum vgroup size, respectively. If a vgroup grows beyond g_{\max} , then we split that vgroup in two smaller ones. When a vgroup shrinks below g_{\min} nodes, we merge it with another vgroup. Parameters g_{\min} and g_{\max} depend on $g = k \cdot \log(N)$.⁴

Group messages. At the group layer, we can view Atum as consisting of a host of robust vgroups. To achieve coordination among vgroups and implement data dissemination, we introduce group messages as a simple communication technique for pairs of vgroups. A group message from vgroup A to vgroup B is a message that all correct nodes in A send to all nodes in B . A node d in B accepts such a message iff d receives this message from the majority of nodes in A , which guarantees correctness of the group message.

Group messages are a central building block of Atum. Two vgroups can exchange group messages only if they know each other's identities, i.e., nodes in vgroup A know the composition of vgroup B and vice versa. We illustrate a group message in Figure 5.3.

⁴The sole purpose of g and k is to better understand Atum's robustness. It is only g_{\min} and g_{\max} that are used as configuration parameters in practice.

5.3.2 Overlay layer

At this layer, Atum maintains an overlay network on top of vgroups that enables the use of group messages – such that vgroups can communicate through gossip. The overlay layer also manages the composition of every vgroup using random walk shuffling. At this layer, each pair of connected vgroups informs each other of any composition change.

The overlay has the form of an H-graph [LS03], in which vgroups correspond to vertices and vgroup connections to edges (see Figure 5.2). An H-graph is a multigraph composed of a constant number of random Hamiltonian cycles. Each vertex thus has two random neighbors for each cycle. This structure is sparse (constant degree), well connected, and has a logarithmic diameter with high probability. Thus, we can apply gossip efficiently on top of this overlay, because messages can permeate rapidly through the whole network. The sparsity of the overlay allows Atum to scale, since every vgroup only has to keep track of a limited (constant) number of neighboring vgroups. A further reason for using this overlay is its decentralized random structure, which is well suited for efficient vgroup sampling using random walks [LS03].

Gossip. We use gossip along the edges of the H-graph, so any two neighboring vgroups can gossip using group messages. We use this technique to disseminate application messages whenever a node invokes a broadcast operation. To transform gossip's probabilistic delivery guarantees into deterministic ones, we have each vgroup gossip at least with neighboring vgroups on a specific cycle of the H-graph.

Random walk shuffling. Like gossip, we also run this protocol along the edges of the overlay. We use random walk shuffling to handle churn, i.e., join and leave operations. Recall that at the group layer we assume that each node has the same constant probability of being faulty (Section 5.3.1). Random walk shuffling guarantees this assumption by assigning joining nodes to vgroups selected uniformly at random from the whole system. As the name of this technique suggests, we use random walks to sample vgroups.

A random walk is an iterative process, where a message is repeatedly relayed across the overlay network. The length of the walks is a system parameter that we denote by $rw1$. At each step of the walk, a vgroup sends a group message to another vgroup using a random incident link of the overlay. After $rw1$ steps, the walk stops at some random vgroup from the network – this is the vgroup which the random walk *selected*. Multiple parameters impact the uniformity of vgroup selection: $rw1$, n/g (i.e, the number of vgroups in the system), and the density of the network (given by hc , the number of H-graph cycles). Intuitively, for uniform selection, a small and dense system needs shorter random walks than a larger, sparser system. In Table 5.1, we summarize the important parameters of Atum.

In order to find proper combinations of Atum parameters (to obtain uniform random vgroup selection), we carry out a simulation. The aim is to derive a guideline that shows the relations between these parameters, so we can properly configure Atum to provide uniform sampling.

Param.	Description	Typical values
hc	Number of H-graph cycles.	2, ..., 12
rwl	Length of random walks.	4, ..., 15
g_{\max}	Maximum vgroup size.	8, 14, 20, ...
g_{\min}	Minimum vgroup size.	$0.5 \cdot g_{\max}$
k	Robustness parameter	3, ..., 7

Table 5.1 – Atum System parameters.

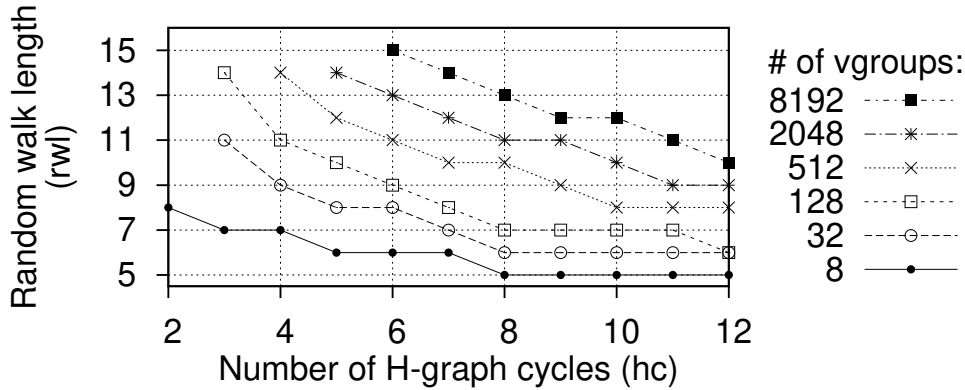


Figure 5.4 – Guideline with optimal rwl and hc system parameters.

Figure 5.4 shows the simulation results. We consider the length of the random walk optimal if Pearson’s χ^2 test with a confidence level of 0.99 cannot distinguish the distribution of the simulated random walks from a truly uniform distribution. The interpretation of this guideline is straightforward; e.g., in a system of roughly 128 vgroups, we set rwl to 9 and hc to 6. In Section 5.6.1, we evaluate experimentally the trade-off between rwl and hc.

Even if new nodes join random vgroups, bugs could lead to faulty nodes accumulating in the same vgroup over time (or an adversary can mount a join-leave attack [AS07]). To counter such a situation, after a node joins or leaves a vgroup, we refresh the composition of that vgroup through a shuffling technique: We exchange all nodes of this vgroup with nodes selected uniformly at random from the whole system. To select a random node from the system, we first use a random walk to select a vgroup, which in turn picks a random node from its composition.

Random walk shuffling ensures that the composition of every vgroup is sampled randomly from the whole system. By keeping vgroups random, we provide a sufficient condition to ensure their robustness. To also ensure efficiency, logarithmic grouping maintains the size of each vgroup logarithmic in system size.

The next section discusses in detail the operations Atum exposes at its interface, as well as the concrete mechanisms used to implement the above concepts such as logarithmic clustering or random walk shuffling.

5.3.3 API operations

Atum exports four basic operations:

- `bootstrap(ownIdentity, params)`,
- `join(contactNode)`,
- `leave()`, and
- `broadcast(message)`.

In addition, our system requires the application to provide two callback functions:

- `deliver(message)`, and
- `forward(message, neighbor)`.

In the following, we describe these operations in detail.

`bootstrap(ownIdentity, params)`

The `bootstrap` operation creates a new instance of Atum that consists of a single vgroup containing only one node – the calling node. Trivially, this vgroup is a neighbor to itself on every cycle of the H-graph. The parameter *ownIdentity* identifies the calling node. It contains the network address (IP address and port) that other nodes can use to join this instance of Atum. The *params* argument specifies system parameters as we present them in Table 5.1.

`join(contactNode)`

As in other BFT systems [KBC⁺00], [RL03], Atum uses a trusted entity to orchestrate the first contact between a joining node and the system. In Atum, any correct participating node can take this role; we call such a node a *contact node*. In practice, the contact node can be a social connection of the joining node, and it is well-known that, without a centralized admission control scheme, a trusted entity is a necessary prerequisite for joining an intrusion-proof system [DLIKA05].

Let *c* be a contact node belonging to vgroup *C*. A `join` operation proceeds as follows. A joining node *j* contacts *c*, which replies with the identities and public keys of nodes in *C*; this is the only step where *j* needs to trust *c*. The joining node then sends a request to be added to the system to all nodes of *C*. After receiving a join request, the nodes in *C* execute an SMR agreement operation [CL02, DS83] to make sure that either all correct nodes of *C* handle the

request or none of them does. This agreement handles the case when j is faulty and sends the join request only to a subset of C .

After agreeing on the join request, C starts the *random walk shuffling* protocol by initiating a random walk. A vgroup D selected by this walk will accommodate j . After the walk finishes, vgroup C sends a group message with the composition of D to j . In the next step, j contacts all nodes in D , these nodes agree on j 's request, update their state, and notify their neighboring vgroups about the new member j . Since we use SMR inside vgroup D , j synchronizes its state with D . The state replicated at each node includes information needed to participate in all protocols, e.g., neighboring vgroup compositions, state of ongoing random walks, or pending join or leave operations.

After vgroup D receives the new node j , random walk shuffling continues by exchanging all nodes of D (including j) with random nodes from the whole system. First, D starts a random walk for each of its nodes to select exchange partners. Let \mathcal{S} denote this set of partners. The next step is exchanging the nodes: (1) each node in D joins the vgroup of its exchange partner, and (2) the partners in \mathcal{S} become members of D .

The last part of the join operation is to check if the size of D exceeds g_{\max} . If it does, we trigger the *logarithmic grouping* protocol. This protocol splits the nodes of D into two equally-sized random subsets – one remains in D , the other forms a new vgroup E . After the split, D starts one random walk for each cycle of the H-graph. Each vgroup selected by such a random walk inserts E between itself and its successor on the corresponding cycle of the H-graph.

leave()

With this operation, a node l sends a request to leave the system to all nodes of its vgroup L . Nodes in L agree on this request, reconfigure to remove l , and inform their neighbors about the reconfiguration. After l leaves, random walk shuffling refreshes the composition of L the same way it does after a new node joins. If this group performs a *merge* (described below), we defer the shuffling until after merging.

If L shrinks below g_{\min} nodes, we trigger logarithmic grouping to *merge* L with a random neighboring vgroup M : All nodes of L join M , and we remove L from the overlay. This removal leaves a “gap” in each cycle of the H-graph. To close these gaps, the predecessor and successor of L on each cycle become neighbors; they receive the information about each other from L . M informs its neighbors about the reconfiguration, shuffles, and splits if necessary.

broadcast(*message*)

This operation allows a node to broadcast a message to all nodes. A broadcast operation comprises two phases. In the first phase, the calling node initiates an SMR operation to do a Byzantine broadcast inside its own vgroup [BSA14, DS83]. In the second phase, Atum uses

gossip to disseminate the message throughout the overlay.

The second phase is customizable, and the application-provided callback `forward` drives the gossip protocol. When a vgroup receives a broadcast message for the first time, Atum delivers this message by calling `deliver`. It then calls `forward` once for each neighbor of that vgroup; this function decides, by returning *true* or *false*, whether to forward a message to a neighbor on the H-graph or not. The default behavior in Atum is to forward broadcast messages to random neighbors, akin to gossip protocols [DGH⁺88].

By modifying the `forward` callback, an application designer can trade-off between message latency, throughput, and fairness. For instance, in latency-sensitive applications, Atum can gossip along *all* H-graph cycles, flooding the system, to disseminate messages fast. For throughput, an application can gossip along a *single* cycle, allowing higher data rates, but increased latency. We experiment with this callback in the evaluation of our data streaming application (Section 5.6.3). We note that an unwise choice of `forward` can break the guarantees of broadcast, for instance, if this callback specifies to *not* forward messages to *any* neighbor.

5.4 Applications

In this section, we illustrate the usage of Atum by designing three applications, which we layer on top of our GCS. We first describe a simple publish/subscribe service, then a file sharing system, and finally a data streaming application.

5.4.1 ASub

Publish/subscribe services are an essential component in cooperative networks and data-center systems alike [CDKR02, EFGK03, GvR13]. ASub is a topic-based publish/subscribe system which relies entirely on the capabilities and API of Atum. We remark that topic-based pub/sub is essentially equivalent to group communication, since the programming interfaces of these two paradigms coincide. The abstraction of a *topic* matches with the abstraction of a *group*, because subscribing to a certain topic involves joining the group dedicated for the said topic; similarly, publishing an event is analogous to broadcasting a message to a group of nodes [Bir93].

Given this equivalence between group communication and pub/sub systems, to build ASub we only need to add a thin layer on top of Atum. Due to space considerations, we do not dwell on the details of this system. Suffice to say that the operations of ASub map directly to the Atum API (Section 5.3.3). Thus, we obtain the following pub/sub operations: `create_topic`, `subscribe`, `unsubscribe`, and `publish` from Atum's bootstrap, `join`, `leave`, and `broadcast`, respectively.

5.4.2 AShare

In this file sharing application, Atum plays a central role by providing the messaging and membership management layer. In AShare, we distinguish between *data*, i.e., file content, and *metadata*, i.e., mapping between files and nodes, file sizes, owners, and file checksums. AShare relies on two protection mechanisms to ensure data availability and authenticity: a novel *randomized replication* scheme to account for high churn, and *integrity checks* to fight file corruption.

AShare stores metadata as soft state, keeping a complete copy at each node, inside a structure called the metadata index.⁵ Whenever a node wants to update the index, it initiates a broadcast, informing every node about the update. We use Atum's broadcast to ensure reliable delivery, so every node correctly receives the broadcast (Section 5.3.3).

Interface and namespace

To add a file with name f in AShare, the owner u calls $\langle \text{PUT}, u, f, c, d \rangle$, where c is the file content and d is the digest of the content. Conversely, $\langle \text{DELETE}, u, f \rangle$ triggers the system to remove all the replicas of f . When nodes want a specific file, they do a $\langle \text{SEARCH}, e \rangle$, where e is the search term, e.g., file or owner name. As a result, SEARCH might yield a file f' previously added by a node u' . To obtain f' , a node calls $\langle \text{GET}, u', f' \rangle$.

The namespace is similar to that of file sharing networks. Every user has its own flat namespace, so we identify files by both their owner and their name (u, f) ; for simplicity, we often omit the owner u when referring to a file. Users have exclusive write access (PUT and DELETE) to their own namespace, and read-only access to foreign namespaces (GET or SEARCH). Being a file *sharing* network, we do not aim at ensuring privacy, but the partitioned namespace restricts malicious activities, given the read-only access. Another advantage of the partitioned namespace is that no updates on the index can ever conflict.

Operations and protection mechanisms

For the sake of availability, AShare replicates every file when the owner calls PUT for that file. In the first step of this operation, the owner u broadcasts a message with the tuple (u, f, d) , making the file available for everyone to read. Upon delivery of this message, every node updates their index to include this new tuple, and then they run a randomized replication algorithm. This algorithm creates multiple replicas of f at random nodes; AShare aims to maintain at least ρ replicas per file, where ρ is a system parameter. In practice, ρ should correspond to a fraction (e.g., 0.1 to 0.3) of the system size. We ensure the availability of every file as long as at least one correct replica exists for each file. ρ replicas thus protect against

⁵An alternative is to use a DHT [SMK⁺01]. This method, however, is fraught with challenges if we want to tolerate arbitrary faults and churn [YKGGK10]. We leave this for future work.

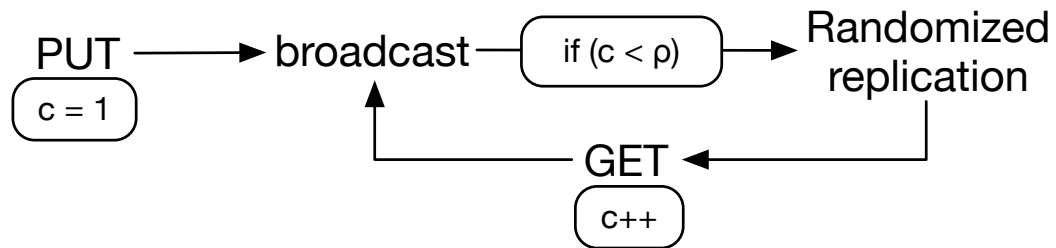


Figure 5.5 – ASHare: A feedback loop triggers the randomized replication algorithm repeatedly. c is the number of replicas for a file.

$\rho - 1$ failures.⁶

Randomized replication. The basic replication algorithm that every node executes is as follows. Given a file f , each node consults its index to compute c , the replica count for f ; if c is smaller than ρ , then every node replicates f with probability $\frac{\rho-c}{n}$, n being the current system size. The outcome of the algorithm is that a random sample of nodes nominate themselves to replicate f , yielding ρ replicas on expectation. To attain ρ copies with certainty, we introduce a feedback loop that triggers the randomization algorithm repeatedly.

Figure 5.5 depicts the feedback loop. To replicate a file (u, f) , a node x simply reads the file by calling $\langle \text{GET}, u, f \rangle$. When GET finishes, x broadcasts the tuple $((u, f), x)$; this broadcast informs every node that x now stores a replica of f . Upon delivering this broadcast, all nodes update their index, and then the feedback loop kicks in: Nodes which do not already store f execute the randomized replication algorithm once more, using the same basic steps we described earlier. The feedback loop deactivates when c (the number of replicas for f) becomes greater or equal to ρ .

During a GET operation, the calling node consults its index to obtain the addresses of all the nodes which store the target file f . The node needs all these addresses because it performs a chunked transfer from multiple nodes at a time (not just from the owner). A problematic situation can appear, however, if some node that stores a replica of this file is faulty and the replica is not consistent with the original file. To solve this issue, we introduce integrity checks.

Integrity checks. This protection mechanism preserves the safety of the service. It allows a node to verify if a replica of a file is authentic, and fights against file corruption that can arise as a result of disk errors [SDG10] or Byzantine faults.

As described earlier, as part of the PUT operation, the owner broadcast a tuple (u, f, d) , containing file identification and digest. We compute the digest using a SHA-2 collision-resistant hash function. The nodes store this digest in the index, and then use it to verify data authenticity.

⁶A failure in this context means that a node holding a file replica misbehaves (e.g. by corrupting replicas), or leaves the system.

Nodes pull files from each other in chunks, i.e., every file has a predetermined number of chunks, established by the owner. Chunks are the units of transfer during GET. This scheme has two benefits: (1) A node can pull file chunks in parallel from all the nodes which replicate that file; (2) digest computation is faster because it can take advantage of multithreading, by computing digests for multiple chunks in parallel. If the integrity check for any chunk fails, then the chunk is pulled from another node. Given this chunked transfer scheme, parameter d in a PUT operation is actually a set of digests, each corresponding to one of the chunks.

We implement the index as a general key-value store using SQLite [sql]. It is useful both to resolve file lookups (by checking the files-to-nodes mapping) and to verify the authenticity of chunks (using digests). If a node detects that its index is corrupted (e.g. due to disk errors or bugs in auxiliary software such as SQLite), it can leave and rejoin the system to obtain a fresh, correct copy of the index.

We implement DELETE using a broadcast, which informs every node to update their index accordingly. If a node stores a replica of the file being deleted, then it also discards the chunks of this replica. SEARCH is straightforward, since every node has the metadata index; we implement this operation on top SQLite's query engine.

5.4.3 AStream

AStream is a streaming application with a two-tier design. Atum represents the first tier, which reliably disseminates stream authentication (digests) from the source node to other nodes. The second tier is a lightweight multicast algorithm which disseminates the actual stream data. Every node uses the digests from the first tier to verify data from the second tier. This second tier has two modules:

A decentralized algorithm to construct a set of spanning trees, and a push-pull multicast scheme to propagate data. We consider these modules interesting in their own right, but due to lack of space we only give a high-level sketch.

Our second tier is inspired from previous solutions on forest-based reliable multicast [CDK⁺03]. We construct a graph (union of trees) with two important properties: (1) It is rooted in the source node (i.e., the broadcasting node), and (2) every node – except the root – has at least one parent which is correct. Intuitively, these two properties ensure that all nodes receive the data stream from the root correctly.

To build a graph with these properties, we leverage the underlying structure of the Atum overlay (see Figure 5.2 for an illustration) as follows. First, we use a deterministic function that every node knows, to pick one of the cycles of the H-graph, denoted w , and a direction d on that cycle (either *left* or *right*). Each node then builds a set of parents of size $f + 1$, chosen randomly from vgroup V , where V is the neighboring vgroup on cycle w and direction d . The nodes which are neighbors with the source choose the source as their single parent – forming the connection to the root. Given the properties of Hamiltonian cycles and the fact that every

vgroup has a majority of correct nodes, this ensures that every node has at least one correct parent, the source node being the root. In addition to this, nodes also select a parent from all other neighboring vgroups, which they may use as shortcuts, in case they are very far from the source node on the selected cycle w .

For disseminating the data, we use a simple, redundant scheme. The root first splits the data in successive chunks, and pushes the first chunk to each of its children. These children, in turn, push this chunk to their children. The algorithm then switches from a push phase to a pull phase, as follows: Each child selects the first parent that pushed a valid chunk, and periodically pulls the subsequent chunks. A node that fails to obtain stream chunks (after receiving the corresponding digests through Atum) tries pulling from another parent. While it is simple, this technique ensures delivery of all data, as at least one parent is always correct.

5.5 Deploying Atum

In this section, we discuss some practical aspects of Atum. We then present our two different implementations of it.

5.5.1 Practical considerations

Message digests. Similar to [CL02], we reduce network bandwidth usage by substituting the content of some messages with their digest. In Atum, a majority of the nodes in any vgroup send the entire group message (Section 5.3.2); the remaining nodes only send a digest of the corresponding message. Since every vgroup has a correct majority of nodes, this strategy ensures that at least one correct node sends the entire message, so we never need to retransmit a message.

Random walk communication. When a vgroup G starts a random walk (Section 5.3.2), the vgroup S selected by this random walk cannot communicate directly with G , because S is a random vgroup from the system, not necessarily a neighbor of G , and thus might not know G 's composition. To deal with this issue, random walks comprise a *backward phase*.

The backward phase of a random walk carries a message from S back to G , relayed by the same vgroups that initially forwarded the random walk. After the backward phase finishes, G and S can start communicating directly, as we piggyback their compositions on the relayed messages.

An alternative solution is what we call *random walk certificates*. They work as follows. At each iteration of a random walk, the forwarding vgroup appends a certificate to the message used to carry out the random walk. This certificate consists of the identity of the chosen neighbor, signed by the forwarding vgroup. When the walk reaches the selected vgroup, it contains a chain of certificates, where each vgroup certifies the identity of the next one. The selected vgroup S can then send a reply directly to the originating vgroup G , with the whole certificate

chain appended. This way, G can verify the identity of S by verifying the certificates in the chain. The advantage of this approach is that a backward phase is not necessary and that vgroups need not keep state of ongoing random walks. Depending on the length of the random walk, however, the certificate chain can become bulky in size (which is linear in $rw1$).

We experiment with both approaches to random walk communication (backward phase and certificates) in our two implementations. The asynchronous implementation uses random walk certificates, since they are conceptually simpler, easier to implement, and incur less total overhead. However, verifying all signatures in a long certificate chain is computationally expensive. Since certificate verification would make it hard for the synchronous implementation to meet its timing deadlines, we opt for the backward phase in the synchronous case.

Bulk RNG for random walks. A random walk of length $rw1$ requires that $rw1$ vgroups generate a random number – each vgroup that forwards the walk to a random neighbor. Distributed random number generation algorithms, however, are expensive [KHG12]. Thus, we generate *all* of the $rw1$ random numbers in bulk at the first iteration of the walk, and we piggyback these numbers on the random walk messages. At each subsequent iteration of the walk, the forwarding vgroup uses one of the $rw1$ random numbers.

Intuitively, one could consider a simpler approach of pre-computing random numbers at each vgroup, and using such a random number pool whenever a random walk needs to be relayed. Interestingly, this approach turns out to be incorrect. A single Byzantine node could bias the random decision by repeatedly triggering operations that consume random numbers from the pool. Thus, to keep our system robust against such an attack, we require that random numbers are not generated before knowing exactly what to use them for.

Randomized message sending. During early experiments with Atum we noticed that the problem of throughput collapse can arise [CGL⁺09]. This can happen when a vgroup has to send one or multiple large group messages. Typically, the size and number of inter-node messages in a group message depend on the size of the communicating vgroups (see Figure 5.3). After the nodes of the sending vgroup generate outgoing messages, they send them in a short burst to the first node of the destination vgroup, then the second node, and so on. In the worst case, there is an upsurge of download bandwidth at each destination node, leading to packet loss.

To address this issue, each sending node randomizes the order of the outgoing messages.

Removing unresponsive nodes. Nodes become unresponsive due to crashes, bugs, network partitions, etc. We use a mechanism similar to `leave` to evict unresponsive nodes. To this end, every node in Atum sends periodic heartbeats to its vgroup peers. If a node fails to send heartbeats, the other nodes in the vgroup eventually agree to evict this node. Eviction proceeds in the same way as a `leave`.

In an asynchronous system, it is impossible to distinguish a failed node from a slow node, so

our heartbeats are coarse-grained, e.g., one every minute. If a node is silent and omits to send many successive heartbeats, amounting to a predefined period of time, then its peers agree to evict the silent node. If an evicted node recovers, it can rejoin the system using `join`. This eviction scheme does not endanger safety, because we evict nodes at a very slow rate. If an attacker wants to break the safety of our system by attacking correct nodes, the attacker would have to mount a persistent barrage of DDoS attacks on many nodes; we believe the resources needed for such an attack outweigh the benefits.

5.5.2 Atum implementations

As explained in Section 5.3, at the design level we make no explicit choice of which SMR algorithm to use inside vgroups. In the first implementation, we choose to use a synchronous algorithm, in particular the Dolev-Strong agreement protocol [DS83] for SMR; synchronous algorithms are significantly simpler to implement, reason about, and debug, compared to their asynchronous counterparts [BSA14, CL02]. To see how this choice impacts performance and to obtain a comprehensive evaluation, we also implement a version of Atum based on the PBFT asynchronous SMR [CL02], combined with an adaptation of the SMART protocol [LAB⁺06] for reconfiguration. We examine the differences between these two implementations in our evaluation (Section 5.6) and further discuss them in the experiences section (Section 5.7). In addition, we also implement the three applications described in Section 5.4.

5.6 Evaluation

We report on our experiments with Atum on Amazon’s EC2 cloud. For the synchronous version (SYNC), we use a single datacenter in Ireland. Due to a high level of redundancy, intra-datacenter networks are synchronous [SHPG12]; indeed, infrastructure services in datacenters often rely on synchrony [CDG⁺08, CLM⁺08, GGL03]. For WAN experiments, we use the asynchronous version (ASYNC) because the network is less predictable. We deploy ASYNC across 8 different regions of the world, located in Europe, Asia, Australia, and America. For both SYNC and ASYNC, each node runs on a separate virtual machine instance of type `micro`, which provides the lowest available CPU and networking performance [amaa].

5.6.1 Base evaluation of Atum

We study here the behavior of the main operations in Atum, so these results pertain to any application layered on top of Atum. Since Atum implementations are user-space libraries, we use the ASub application to carry out these base experiments. We deploy both SYNC and ASYNC and address four important questions: (1) How fast can the Atum system grow? (2) What continuous churn rate can it sustain? (3) How fast does Atum disseminate messages (a) in a failure-free scenario (b) and in presence of Byzantine nodes?

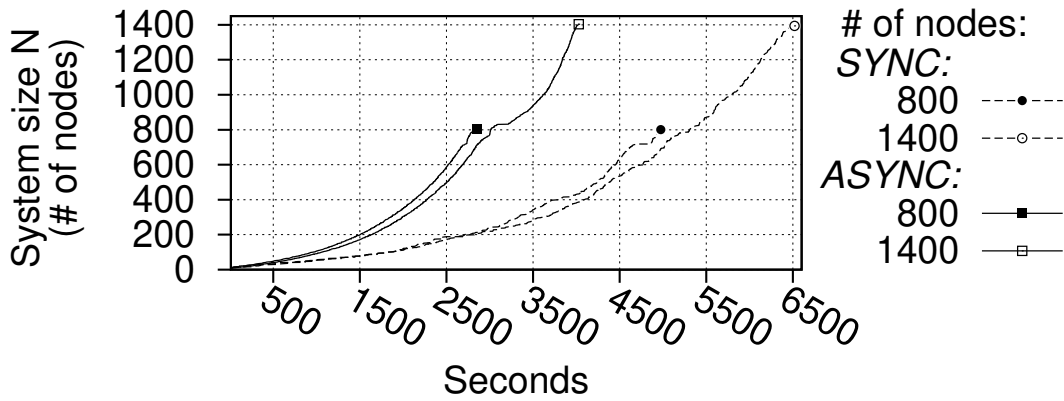


Figure 5.6 – Growth speed for systems with up to 1400 nodes.

System growth speed

We first consider the join throughput; this may reflect, for example, the arrival rate of new subscribers in ASub. We use different configurations of (hc, rwl) , depending on the target system size, according to our guideline in Figure 5.4. E.g., for a system with 800 nodes in roughly 120 vgroups, $(hc, rwl) = (5, 10)$. For the SYNC system we use rounds of 1 second, and we evaluate both versions using systems of 800 and 1400 nodes.

As Figure 5.6 shows, if we configure a system for a smaller maximum size, then the system can grow slightly faster. This is because larger systems require larger values for rwl , which increases the cost of adding nodes. As the system grows, however, it is able to handle faster rates of node arrival, resulting in an exponential growth; our flexible overlay allows the system to run multiple join operations concurrently, all of which execute within a confined (randomly selected) part of the network. During these experiments, we do not observe any scalability bottlenecks. We expect Atum to continue to exhibit this behavior (good scalability and exponential growth speed) in systems well beyond 1400 nodes, so, in the interest of time and budget, we choose to not experiment further.

Note the glitch around second 3000. The growth rate drops slightly due to temporary asynchrony, after which many nodes join in a burst and the system continues to grow normally. The short plateau after the burst is caused by the delay in creating and booting Amazon instances.

Churn tolerance

Next, we provoke continuous churn by constantly removing and re-joining nodes, for systems of up to 800 nodes. As we show in Figure 5.7, SYNC can churn up to 18% of all nodes every minute, and ASYNC reaches 22.5%. The nodes have an average session time between 5 and 6 minutes.

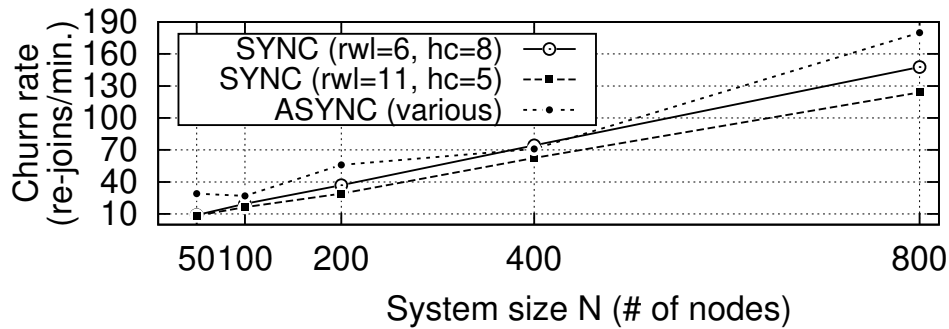


Figure 5.7 – Maximal tolerated churn rates in systems of size 50, 100, 200, 400 and 800 nodes.

We use SYNC to also evaluate how the choice of overlay parameters affect Atum’s behavior under churn. The relevant parameters are random walk length ($rw1$) and number of H-graph cycles (hc). We use two combinations of ($rw1$, hc): (6, 8) and (11, 5). The intuition is that random walks are heavily used during churn, so we expect that a smaller $rw1$ allows higher churn rates. Figure 5.7 confirms this intuition. The decrease in $rw1$ does not translate, however, to a proportional increase of churn rate, because other sub-protocols also affect this rate, e.g., random number generation or SMR inside vgroups. Since the behavior of ASYNC is less predictable, this effect, although present, is less prominent for ASYNC, and we use a different configuration for each system size according to our configuration guideline.

As our configuration guideline (Figure 5.4) shows, if we decrease $rw1$ (y-axis), we have to increase hc (x-axis) to preserve the random walks’ uniform sampling property. A bigger hc means that groups have more neighbors, so nodes keep more state, which leads to bulkier state transfers. Going from $hc=5$ to 8, however, turns out to have a smaller impact on the churn rate than the change of $rw1$.

Group communication latency

In this experiment, we instantiate a system and then disseminate 800 messages of length 10 to 100 bytes (comparable to Twitter messages). We experiment with system sizes of 200, 400, and 800 nodes in a failure-free case. To also evaluate Atum in the presence of faults, we use 800 correct nodes and subsequently add 50 (5.8%) nodes with injected faults.

To simulate arbitrary behavior of Byzantine nodes in SYNC, we modify their algorithm such that they do not participate in any protocol except: (1) they send heartbeats, to prevent being evicted from the system (see Section 5.5.1); and (2) they pretend not to receive heartbeats from correct nodes, and periodically propose to evict all correct nodes from their vgroup. A Byzantine node has no incentive to send spoofed or corrupted messages while the majority of the nodes in its vgroup is correct; the recipient of such messages would discard them. To set the system parameters, we use the configuration guideline (Section 5.3.2), and we use rounds of 1.5 seconds. For ASYNC, faulty nodes have no incentive to send corrupted messages, and therefore stay quiet.

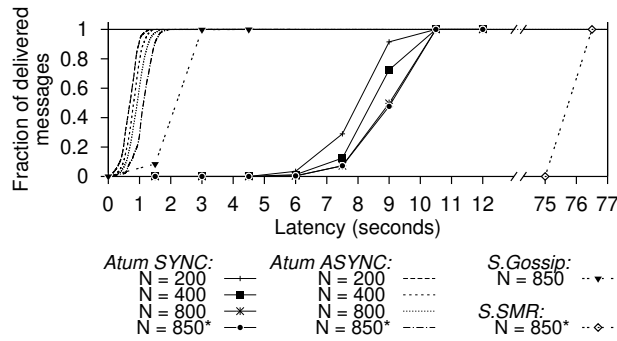


Figure 5.8 – Group communication latency: Comparison between gossip, Atum, and SMR. We tag with * the systems with 50 faults.

We depict a CDF of the obtained latencies in Figure 5.8. For all scenarios with SYNC, the latency has an upper bound of 8 rounds (12 seconds). The two phases of broadcast contribute independently to this latency. Nodes in the same vgroup with the publisher deliver a message immediately after the first phase, and nodes in other vgroups deliver it in the second phase (Section 5.3.3). We normalize the results to correspond to the expected latency of the first phase, which is 4 rounds in these experiments. The actual latency might differ by up to 2 rounds, depending on the size of the publisher’s vgroup. Since faulty nodes do not reach majority in any vgroup, they do not affect correct nodes. Thus, Atum suffers no performance decay despite 5.8% faults, and the latency remains unchanged.

Figure 5.8 also shows how SYNC compares with the two approaches it combines: synchronous SMR and gossip. The first baseline is a simulation of a classic round-based crash tolerant gossip protocol [DGH⁺88], with no failures. Every node has a global membership view and in every round exchanges messages with random nodes. To ensure fair comparison, we set the fanout of this gossip protocol (i.e., the number of message exchanges per round) to the size of the view of a Atum node (this is a loose upper bound on the Atum fanout), and rounds to 1.5 second. As Figure 5.8 reveals, the latency penalty in Atum corresponds roughly to the latency of the SMR protocol in the first phase of broadcast (which is 4 rounds), with minimal additional overhead. This is the price we pay for Byzantine fault tolerance. The second baseline is the Byzantine agreement protocol [DS83] that we use to implement SMR in SYNC, when we scale it out to the whole system. The latency for this protocol is $f + 1$ rounds (1.5 seconds each), where f is the number of tolerated faults.

Latencies for ASYNC are much lower, since there are no synchronous rounds, and so nodes do not need to coordinate their steps at a conservative rate. In contrast to SYNC, however, the tail latency reaches 105.5s, with less than 0.01% notifications delayed by more than 5s. With ASYNC, a small number of temporarily slow nodes might deliver notifications late, without affecting other nodes. To compensate for the lower fault tolerance of ASYNC ($\lfloor (g - 1)/3 \rfloor$ instead of $\lfloor (g - 1)/2 \rfloor$), we increase the robustness parameter k to 7, which results in a latency increase due to larger vgroups.

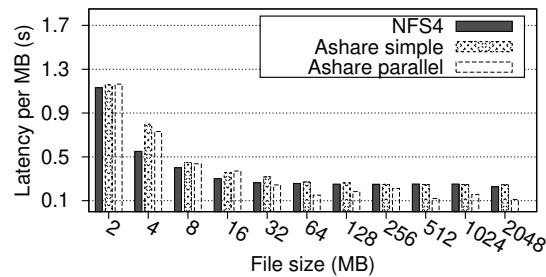


Figure 5.9 – AShare: Read performance (latency per MB). We normalize the result to file size.

Failure rates observed in practice are about 0.5% nodes per day in a datacenter, according to Barroso et al. [BCH13]. In this experiment, we tolerate 5.8% faults thanks to our logarithmic grouping and random walk shuffling schemes. In fact, the number of faults that Atum tolerates increases with system size. This is intuitive, given that larger systems imply larger vgroups (vgroups are roughly logarithmic in system size), so each vgroup can handle more faults.

We obtain the results hitherto using ASub, since this application maps exactly to the group communication API of Atum. Nevertheless, these results evaluate the basic operations of Atum, so they apply to all applications built on Atum, including AShare and AStream. In the following sections, we evaluate particular metrics for these two applications; this evaluation is orthogonal to the underlying GCS and independent of the group communication performance.

5.6.2 Evaluating AShare

We first evaluate the performance of GET in failure-free runs. This operation is equivalent with reading an entire file in a typical distributed filesystem, so we use NFS4 as baseline. By default, NFS4 has no fault-tolerance guarantees and is the standard solution for accessing files across the network. Results show that we provide comparable performance with NFS4, while offering stronger guarantees.

Figure 5.9 shows our results. We normalize the read latency to file size, so the y-axis plots latency/MB, using files from 2MB to 2GB. We consider three cases: (1) NFS4, where a client reads from a server; (2) AShare *simple*, where a node GETs files replicated by another node and the files have a single chunk – this is for fair a comparison with NFS4; and (3) AShare *parallel*, where a node reads files replicated by two other nodes and each file has 10 chunks. As we can see, the normalized read latency decreases as file size increases, because the constant overhead for transfer initiation (e.g., handshakes, TCP slow-start [APB09]) amortizes as transfer time grows. While the AShare simple execution can match the performance of NFS4 for larger files, we observe that the parallel execution outperforms NFS4 by up to 100% for files over 512MB. We attribute this gain to the use of parallel pulling and multithreaded digest computation.

We also study how Byzantine nodes impact GET latency. A Byzantine node in this scenario corrupts all the replicas it stores. We analyze the read latency in two scenarios – a 50-node

system with 500 files, and a 100-node system with 1000 files. In both scenarios we set $\rho = 8$, so each file has a minimum of 8 replicas, and 7 random nodes are Byzantine. Every file consists of 10 chunks, with a fixed size of 1MB.

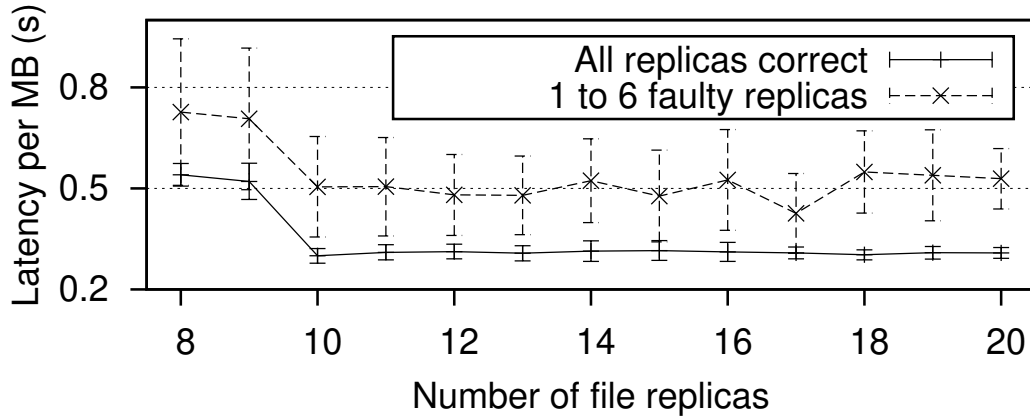


Figure 5.10 – AShare: Impact of Byzantine nodes on read latency. Experiment with 50 nodes (7 Byzantine) and 500 files.

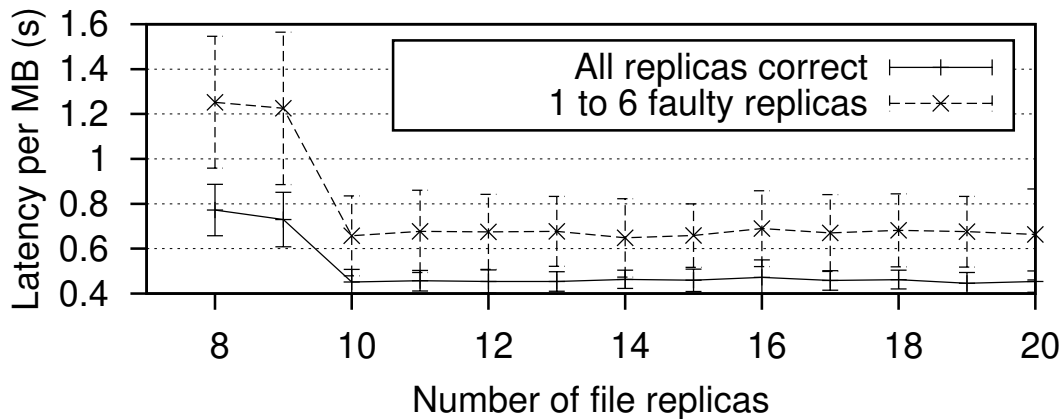


Figure 5.11 – AShare: Impact of Byzantine nodes on read latency. Experiment with 100 nodes (7 Byzantine) and 1000 files.

Figure 5.10 conveys AShare’s resilience to corrupted replicas in the 50-node system. For moderately-replicated files, with 8 or 9 replicas, the read latency increases by up to 3x. This is expected, given that the majority of the pulled chunks are corrupted and have to be re-pulled from a correct node. We also observe that the positive effect of having multiple replicas per file diminishes. In the ideal configuration, the number of chunks equals the number of replicas of a file, such that each chunk can be pulled from a separate node and verified in parallel. In this configuration we can strike a balance between storage overhead (replicas count) and read latency. This can be seen in our results for files with 10 replicas. We draw similar conclusions from the results of the 100-node experiment in Figure 5.11.

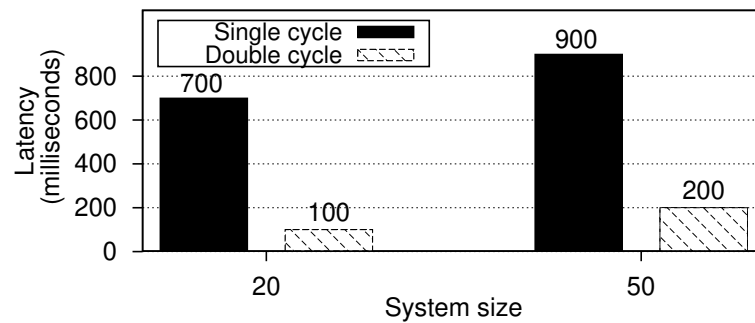


Figure 5.12 – AStream: Latency for 1MB/s data stream.

We also used the Grid'5000 experimental platform for AShare evaluation, both for the failure-free and Byzantine scenario. Compared to EC2, we experimented on better machines (Xeon or Opteron CPUs, more memory) on a network with similar properties. We omit the results for brevity, as they are consistent with the results on EC2.

5.6.3 Evaluating AStream

To evaluate AStream, we consider a 1MB/s stream, which is an adequate rate for live video. As discussed in Section 5.3.3, the forward callback allows applications to customize the way Atum disseminates data in the second phase of broadcast. Specifically, applications like ASub would favor latency and flood the system by gossiping on all the H-graph cycles. In AStream, latency is not critical, so we customize the forward callback to use either one (*Single*) or two (*Double*) H-graph cycles. For SYNC, we set the round duration to 1 second. We run experiments with 20 and 50 nodes. In parallel with the first tier, the second tier transfers the data as soon as Atum delivers the digests.

In Figure 5.12 we plot the latency of the second tier of AStream. As we show, changing the forward function has an impact on dissemination. As expected, if we use more cycles to disseminate metadata, latency decreases. Since we use a lightweight multicast protocol in the second tier, this has a small impact on latency. For instance, in the 20-node system, Single scenario, the second tier takes .7 seconds; the total latency is 5.7 seconds with SYNC (which incurs 5s latency) or 1.7 seconds with ASYNC (which incurs 1s).

5.7 Experiences and Lessons Learned

A big challenge we faced concerns the fundamental trade-off between flexibility and robustness. On the one hand, Atum is designed to be flexible and adapt to high churn with ease: It resizes, merges and splits vgroups as befitting. On the other hand, to uphold robustness, Atum is also designed to restrict how vgroups evolve, placing bounds on their size, and composing them via random sampling. To convey how this trade-off manifests in practice, we strain Atum

by joining nodes at an overwhelming rate; this generates many concurrent shuffle operations, and suppresses some node exchanges, because the chosen exchange partner already participates in another exchange. In our experiments (Section 5.6.1) we join nodes at a rate of 8% of the system size each minute. Figure 5.13 shows what happens when we intensify this rate to 20% and 24%: Higher growth rates suppress more node exchanges (penalizing robustness), but the system grows faster (is more flexible).

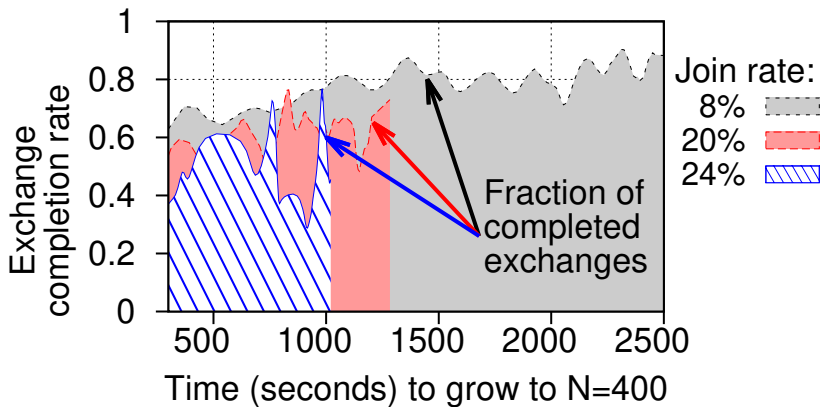


Figure 5.13 – As a system grows faster, the quality of random vgroup composition suffers due to suppressed exchanges.

Another hard challenge was the interplay between the SMR algorithm inside vgroups and the distributed protocols running among vgroups. The two most salient issues here were the following. First, SMR reconfiguration by itself is a tricky business [BSA14, OO14]. The shuffle operation in Atum, however, involves multiple vgroups concurrently reconfiguring by exchanging nodes among themselves. Complications with this include deadlocks, missed operations, duplicate membership (nodes belonging to two different vgroups), dangling membership (nodes being left out of a vgroup), or other inconsistencies. Second, the H-graph overlay we use is decentralized and random, where every node has only a local view; this makes the split operation particularly intricate because it involves orchestration among multiple vgroups, on all cycles of the graph.

These challenges compelled us to simplify the implementation at all levels. For this reason, we initially considered a synchronous SMR [DS83], and implemented SYNC in 20K LOC in C. This decision is reasonable for highly-redundant networks (such as inside a datacenter availability zone [SHPG12]), but is not realistic for large, dynamic networks, where the round size has to be very conservative. Since we wanted a comprehensive evaluation, including on WAN, we decided to also implement ASYNC, based on asynchronous SMR, i.e., assuming eventual synchrony [CL02]. This version is more complex, but we used a high-level language (8K LOC in Python), which helped us deal with the complexity.

As our results show, ASYNC outperforms SYNC. On the other hand, SYNC brings multiple benefits: It has predictable performance, is simpler to implement and reason about (due to

the round-based model), and is thus less prone to bugs; moreover, SYNC was an important stepping stone to help us understand the complex interactions in Atum.

5.8 Conclusions

This chapter reports on our experiences with designing and building Atum, a general-purpose group communication middleware for a large, dynamic, and hostile network. At the heart of Atum lies the novel concept of volatile groups, i.e., small, dynamic, yet robust clusters of nodes. Specifically, Atum applies state machine replication at small-scale, inside each vgroup, and uses gossip to disseminate data among vgroups. We ensure that vgroups are robust and efficient by employing two techniques, namely random walk shuffling and logarithmic grouping.

We experimented with two Atum implementations – one synchronous and one asynchronous (eventually synchronous). We used Atum as a reliable core to build three applications: ASub, a publish/subscribe service, AShare, a file sharing system, and AStream, a data streaming platform. Using these applications, we verified experimentally the properties of our system: Our findings indicate that Atum tolerates arbitrary faults even in a large-scale, high-churn network.

6 Related Work

This chapter reviews selected publications related to the research presented in this dissertation. We first handle literature focused on the specific problem of transferring digital assets, and then discuss work that concerns the more general problem of group communication and some of its applications.

6.1 Asset Transfer

In this section we discuss various solutions of the asset transfer problem (discussed in chapters 2 and 3), both for the private (permissioned) and public (permissionless) setting. Our high-level solution is suitable for both a private (permissioned) and public (permissionless) setting, as it builds on high-level abstractions of read-write memory or secure broadcast.

Many systems address the problem of asset transfers, be they for a private or public setting. In the private case [ABB⁺18, Hea16, KJW⁺18], the algorithm assumes the existence of some explicit (e.g., certificate authority) or implicit (e.g., physical connection to a system) way of controlling access to the system. This assumption prevents from having to deal with potential Sybil attacks [Dou02], where an adversary takes control over a system by overwhelming it with many artificially generated identities that the adversary controls. Distributed systems for the public environment [AMN⁺16, DSW16, GHM⁺17, KKJG⁺18, Nak08, TR18] can be joined by anyone. To prevent a malicious adversary from overtaking the system, these systems rely on proof-of-work [Nak08], proof-of-stake [BGM16], or other techniques designed to protect from Sybil attacks.

The above-mentioned solutions, whether for the permissionless or the permissioned environment, seek to solve consensus. They must inevitably rely on synchrony assumptions or randomization. By sidestepping consensus, we can provide a deterministic and completely asynchronous implementation.

It is worth noting that many of those solutions allow for more than just transfers, and support richer operations on the system state—so-called smart contracts. In this work we focused on

the original asset transfer problem, as defined by Nakamoto [Nak08], and we did not address smart contracts, for certain forms of which consensus is indeed necessary. However, our approach allows for arbitrary operations, if those operations affect groups of the participants that can solve consensus among themselves. Potential safety or liveness violations of those operations (in case this group gets subverted by the adversary) are confined to the group and do not affect the rest of the system.

In the blockchain ecosystem, much work has been devoted to replacing a totally ordered transaction ledger by a directed acyclic graph (DAG) representing the relations between transfers. DAG-based systems include Nano [LeM18], Corda [Hea16], or Vegvisir [KJW⁺18]. While replacing a linear blockchain by a DAG, these systems still do rely on consensus in some form.

In our case, the dependencies between transfers can also be represented by a DAG, but we do not resort to solving consensus to build the DAG, nor do we use the DAG to solve consensus. More precisely, we can regard each account as having an individual history. Each such history is managed by the corresponding account owner without depending on a global view of the system. Histories are loosely coupled through a causality relation established by dependencies among transfers.

One could also relate our asset transfer object (Section 2.2.2) to conflict-free replicated data types (CRDTs) [SPBZ11]. As in CRDTs, we keep the state of the object consistent across replicas while updates are being applied concurrently. Accounts of correct processes never experience conflicting updates, as only one process—the owner—is allowed to issue operations that can potentially produce conflicts. This, however, never happens concurrently, as processes are assumed to be sequential. (For shared accounts, processes have to coordinate before performing any update.) From the point of view of Shapiro et al. [SPBZ11], the asset transfer object is strongly eventually consistent.

Pedone and Schiper [PS02] already discussed that transferring assets is possible in an asynchronous way, relying only on broadcast. This insight has also been used by Duan et al. [DRZ18] in Byzantine tolerant storage protocols. Gupta [Gup16] also presented algorithms for financial transfers based on ideas similar to the ones underlying our asset transfer protocol and the broadcast abstraction it uses [MMR97, MR97b]. However, to the best of our knowledge, we are the first to formally define the asset transfer problem as a shared object type, study its consensus number, and propose algorithms building on top of standard shared memory / broadcast abstractions.

6.2 Group Communication

In this section we discuss existing work related to our contributions in broadcast, group communication and related applications, mostly concerning chapters 4 and 5. We review some of the work which intersect with our efforts and highlight the differences. The reviewed

work spans multiple research areas, including broadcast abstractions, gossip, robust overlay networks, membership sampling, and storage systems, which we discuss in the following.

6.2.1 Broadcast abstractions

Our secure broadcast abstraction is based on Bracha’s Byzantine reliable broadcast [Bra87], both in terms of abstraction and implementation. At both levels, however, we generalize it. In the abstraction, we introduce the notion of ϵ -security, allowing some properties to be violated with probability ϵ . At the level of implementation, we build on a similar basic idea as Bracha’s broadcast (referred to as double-echo broadcast in [CGR11]). By leveraging the possibility of failing with a low probability, we achieve lower complexity and thus higher scalability compared to Bracha’s original implementation. Intuitively, Atum provides similar guarantees for each invocation of its broadcast primitive, even though we did not formally analyze it in terms of ϵ -security.

The variant of secure broadcast used by our asset transfer algorithm has been referred to by Malkhi et al. as secure reliable multicast [MMR97, MMR00, MR97b]. Unlike to Byzantine reliable broadcast, it considers multiple senders and multiple messages and provides some ordering guarantees (expressed either through delivery order [MR97b] or sequence numbers [MMR97]). The implementations proposed so far either deterministically rely on quorums, which quickly becomes a bottleneck for bigger system sizes, or lack thorough analysis in the probabilistic case.

6.2.2 Gossip

Our broadcast solution (both in Ready Broadcast and in Atum) is based on gossip, first introduced by Demers et al. to disseminate updates in replicated databases [DGH⁺88]. Gossip-based systems typically exploit randomness to bypass failures and ensure robust dissemination. In [MMR99], Malkhi et al. studied the diffusion problem in a Byzantine environment, where a message that is initially known by a small number of source nodes must be disseminated to and accepted by all correct nodes. They proposed a class of *direct verification*¹ protocols that solve the diffusion problem. They proved lower bounds on the dissemination time of this class of protocols, rendering the protocols impractical at large-scale. Minsky and Schneider proposed *path verification* protocols [MS03] for the Byzantine diffusion problem, generalizing direct verification and circumventing the lower bounds of [MMR99]. However, the dissemination time of path verification protocols is linear in the number of tolerated failures. Scalability is thus still limited if the fraction of Byzantine nodes is constant.

FlightPath [LCM⁺08] is a powerful P2P streaming system. It is robust towards rational and Byzantine nodes, and it tolerates high churn rates. In FlightPath, the focus is on streaming data efficiently from a designated source node, using gossip and a centralized tracker. In our

¹The term “direct verification” was introduced later by Minsky and Schneider [MS03]

work we also leverage gossip; in addition, Atum is a completely decentralized, general-purpose GCS, it does not rely on a central tracker, and it allows *any* node to broadcast.

6.2.3 Robust overlay networks

S-Fireflies [DHR07] is an effective self-stabilizing overlay network suitable for data dissemination. It is robust to Byzantine faults and churn-tolerant. It creates random permutations of the system nodes and uses them to pick the neighbors for each node. A difference between S-Fireflies and Atum is that the former relies on global knowledge of all nodes at all times, meaning that every node of the system is aware of every other node.²

Vicinity [VvS13] is a protocol for constructing and maintaining an overlay network, which investigates the importance of randomness in large-scale P2P networks. The analysis around this protocol shows that randomness must be complemented with structure (determinism) for effective large-scale P2P networks. In Atum, we also leverage non-determinism (shuffling) alongside determinism (SMR) to scatter faulty nodes and handle churn in volatile groups.

DHTs such as Chord [SMK⁺01] or Tapestry [ZHS⁺04] provide $O(\log N)$ lookup time and are commonly used for routing in storage system. DHTs can be adapted to handle Byzantine faults [CDG⁺02, FSY05, GFG⁺09, RLS02], and the problem of churn has also been addressed in [LYL⁺06, RGRK04]. Tentative lookup schemes that are both secure and churn-friendly are given in [YK GK10], where session times as low as 10 minutes are theoretically explored. In Atum, we focus on general-purpose group communication – instead of lookup – and we use the novel concept of volatile groups to allow even shorter session times.

6.2.4 Membership sampling

Membership services are an important middleware for data dissemination [FC97, HS00]. These services often rely on the non-determinism of a sampling protocol to obtain random connections with other peers and ensure low-latency and robust dissemination. In Atum, our sampling scheme is based on random walks over a well-connected H-graph. Other systems, such as RaWMS [BYFK08] or the wormhole-based peer sampling service (WPSS) [RDJ13], also leverage random walks to implement efficient sampling. Brahms [BGK⁺09] is a Byzantine-tolerant membership sampling algorithm that tolerates churn. It is based on gossip and provides a close-to-uniform sample of the processes across the whole system. In contrast with these systems, Atum provides a complete solution for data dissemination, which also handles high churn rates and Byzantine failures.

Distributed clustering techniques seek to group the processes of a system into clusters, sometimes called shards or quorums, of size $O(\log N)$ [AS09, KLST11, Sch05]. This line of work has various goals, but similar to our efforts, they also aim for scalable solutions. The overarching

²We also assume global knowledge in one of our applications (AShare, Section 5.4.2), but this assumption is not inherent in Atum.

principle in clustering techniques is similar to our use of samples: build each cluster in a provably random manner so that the adversary cannot dominate any single cluster, i.e., contain Byzantine processes within their cluster. Samples in our solution are private and individual per-process, in contrast to clusters which are typically public and global for the whole system. To the best of our knowledge, we are also the first to explore groups of logarithmic size towards obtaining a highly-scalable secure broadcast algorithm.

6.2.5 Storage

Scatter [GBKA11] is a distributed key-value store with linearizable operations. As Atum, Scatter also partitions the system into self-organizing, dynamic groups of nodes, and is churn-tolerant. Scatter focuses on performance and strong consistency at large scale, while in Atum, our main objective is large-scale BFT. Inside groups, Scatter relies on Paxos; across groups, it achieves coordination using a 2PC-based protocol. Atum ensures stronger fault-tolerance guarantees inside groups (by using a BFT agreement [DS83]) and coordinates groups using a scalable gossip scheme.

BFS [CL02], Farsite [ABC⁺02], Pond [REG⁺03], and Rosebud [RL03] are file storage systems that use PBFT [CL02]. In BFS, the replicas running PBFT also store the data objects, so this system fully replicates data across all the nodes and cannot scale well. Pond, Rosebud, and Farsite achieve scalability by separating the BFT mechanism from the storage subsystem. They use BFT quorums to agree on the operations that are performed, and the storage nodes perform these operations. The BFT and storage subsystems can scale independently in this case. Although these four systems assume a dynamic environment, only Rosebud provides details on this concern.

In Rosebud, a group of BFT replicas agree on the system configuration and monitor all the nodes in the system; this group periodically – once per *epoch* – propagates new configurations. The churn rate in Rosebud thus depends on the length of epochs. This system has an evaluation with epoch duration of a few hours; shorter epochs are possible but are not considered. Atum can cope with session times in the order of minutes (Section 5.6.1). Unfortunately, we were not able to find a Rosebud implementation to use for comparison.

7 Conclusions

This thesis discussed Byzantine fault-tolerance in the context of large-scale systems, focusing on specific applications. The common denominator of these applications is their limited reliance on a solution of the consensus problem. We showed that consensus, often a scalability bottleneck in practical systems, can either be confined to a small part of the system or avoided altogether. In particular, we studied the problem of transferring assets between accounts, we presented a scalable secure broadcast algorithm, and we provided a middleware system for organizing nodes in an overlay network that masks Byzantine faults and supports high churn.

First, we addressed the the asset transfer problem, where assets are being transferred between accounts, while each account has a single owner. While many existing solutions to this problem are based on a solution of consensus at least in some form, and thus fundamentally rely on either synchrony assumptions or randomization (often both), we proved that asset transfer can be solved both deterministically and asynchronously at the same time.

To this end, we examined the asset transfer problem in the context of shared memory, providing its precise formal definition as a sequential object type. We proved that this object type has consensus number 1 by devising an algorithm implementing asset transfer in read-write shared memory using an atomic snapshot object (known to be of consensus number 1).

Furthermore, we extended our theoretical result to the case where an account may have more than one single owner. We proved that a sequential object type that supports multiple owners per account, each of which is able to atomically transfer money from the account, has consensus number k , if each account is shared by at most k owners. In practice, this implies that in a potentially very big system, not all participants need to solve consensus. Instead, it is sufficient if only the (potentially much smaller) set of co-owners of an account can agree among themselves.

Next, we studied the the asset transfer problem in the Byzantine message passing model. We provided a consensusless asset transfer algorithm that is based on a secure broadcast abstraction with weaker than FIFO ordering guarantees. Our algorithm remains deterministic

and asynchronous, using the secure broadcast primitive as a black box.

However, deterministic implementations of secure broadcast suffer from limited scalability, as they involve processes communicating with quorums of other processes, leading to at least linear per-process complexity. We thus also provided a relaxed version of secure broadcast, called probabilistic secure broadcast, in which some of properties are allowed to be violated with a bounded probability. Leveraging this relaxed specification allowed us to circumvent the scalability bottlenecks of deterministic secure broadcast implementations while keeping the probability of failure arbitrarily small. We devised a randomized algorithm implementing secure broadcast that, instead of having processes communicate with quorums of other processes, only communicate with randomly picked samples of processes. A sample being representative of the whole system, a process can infer the state of the system by only observing the state of a sample. The main advantage of using samples instead of quorums is that while quorum sizes are linear in the system size, samples only need to be of logarithmic size.

We also explored the idea of logarithmically sized samples in a very different way in our Atum system. The overarching goal, however, remains the same—scalability. Atum partitions the whole system into groups of logarithmic size, the composition of which is sampled uniformly at random. This ensures, with high probability, that a Byzantine adversary is unable to make malicious processes over-represented in any of these groups. Executing a Byzantine fault-tolerant agreement protocol inside each group, Atum masks malicious behavior of faulty nodes inside each group and thus a group can function as one reliable entity. We achieved scalability by confining the Byzantine agreement protocol to a small group instead of executing it among all processes in the system. We demonstrated the usefulness of this system by implementing three applications that can easily be built on top of Atum: a publish-subscribe system, a file sharing service, and a data streaming system.

While consensus remains a central problem of distributed computing, in this thesis we argued that it is, sometimes surprisingly, not necessary to solve consensus in order to implement many applications of practical relevance. Even in some cases where consensus indeed is required, it may be only needed in a localized way by a small number of processes, without interfering with the rest of the system, greatly improving the system's scalability.

Bibliography

- [AAD⁺93] Yehuda Afek, Hagit Attiya, Danny Dolev, Eli Gafni, Michael Merritt, and Nir Shavit. Atomic snapshots of shared memory. *Journal of the ACM*, 40(4), 1993.
- [ABB⁺18] Elli Androulaki, Artem Barger, Vita Bortnikov, Christian Cachin, Konstantinos Christidis, Angelo De Caro, David Enyeart, Christopher Ferris, Gennady Laventman, Yacov Manevich, Srinivasan Muralidharan, Chet Murthy, Binh Nguyen, Manish Sethi, Gari Singh, Keith Smith, Alessandro Sorniotti, Chrysoula Stathakopoulou, Marko Vukolic, Sharon Weed Cocco, and Jason Yellick. Hyperledger fabric: a distributed operating system for permissioned blockchains. In *EuroSys*, 2018.
- [ABC⁺02] Atul Adya, William J Bolosky, Miguel Castro, Gerald Cermak, Ronnie Chaiken, John R Douceur, Jon Howell, Jacob R Lorch, Marvin Theimer, and Roger P Wattenhofer. FARSITE: Federated, available, and reliable storage for an incompletely trusted environment. *ACM SIGOPS Operating Systems Review*, 36(SI), 2002.
- [ABND95] Hagit Attiya, Amotz Bar-Noy, and Danny Dolev. Sharing memory robustly in message-passing systems. *JACM*, 42(1), 1995.
- [ACMP11] Atul Adya, Gregory Cooper, Daniel Myers, and Michael Piatek. Thialfi: a Client Notification Service for Internet-Scale Applications. In *SOSP*, 2011.
- [AGK⁺15] Pierre-Louis Aublin, Rachid Guerraoui, Nikola Knežević, Vivien Quéma, and Marko Vukolić. The next 700 bft protocols. *ACM Trans. Comput. Syst.*, 32(4), January 2015.
- [AGM⁺17] Ittai Abraham, Guy Gueta, Dahlia Malkhi, Lorenzo Alvisi, Rama Kotla, and Jean-Philippe Martin. Revisiting fast practical byzantine fault tolerance. *arXiv preprint arXiv:1712.01367*, 2017.
- [AGMS18] Karolos Antoniadis, Rachid Guerraoui, Dahlia Malkhi, and Dragos-Adrian Seredinschi. State Machine Replication is More Expensive Than Consensus. In *DISC*, 2018.
- [amaa] <https://aws.amazon.com/ec2/instance-types/>.

Bibliography

- [amab] Amazon S3 Availability Event: July 20, 2008. <http://status.aws.amazon.com/s3-20080720.html>.
- [AMN⁺16] Ittai Abraham, Dahlia Malkhi, Kartik Nayak, Ling Ren, and Alexander Spiegelman. Solida: A blockchain protocol based on reconfigurable byzantine consensus. *CoRR*, abs/1612.02916, 2016.
- [AMQ13] Pierre-Louis Aublin, Sonia Ben Mokhtar, and Vivien Quéma. Rbft: Redundant byzantine fault tolerance. In *ICDCS*, 2013.
- [AO09] Daron Acemoglu and Asu Ozdaglar. 6.207/14.15: Networks - lecture 4: Erdős–rényi graphs and phase transitions. <https://economics.mit.edu/files/4622>, 2009.
- [APB09] M. Allman, V. Paxson, and E. Blanton. TCP Congestion Control. RFC 5681 (Draft Standard), September 2009.
- [AS07] Baruch Awerbuch and Christian Scheideler. Towards Scalable and Robust Overlay Networks. *IPTPS*, 2007.
- [AS09] Baruch Awerbuch and Christian Scheideler. Towards a Scalable and Robust DHT. *Theory of Computing Systems*, 45(2), 2009.
- [AW94] Hagit Attiya and Jennifer L. Welch. Sequential consistency versus linearizability. *ACM TOCS*, 12(2), 1994.
- [BCH13] Luiz André Barroso, Jimmy Clidaras, and Urs Hölzle. The datacenter as a computer: an introduction to the design of warehouse-scale machines. *Synthesis Lectures on Computer Architecture*, 8(3), 2013.
- [BGK⁺09] Edward Bortnikov, Maxim Gurevich, Idit Keidar, Gabriel Kliot, and Alexander Shraer. Brahms: Byzantine resilient random membership sampling. *Computer Networks*, 53(13), 2009. Gossiping in Distributed Systems.
- [BGM16] Iddo Bentov, Ariel Gabizon, and Alex Mizrahi. Cryptocurrencies without proof of work. In *International Conference on Financial Cryptography and Data Security*. Springer, 2016.
- [BGP89] Piotr Berman, Juan A. Garay, and Kenneth J. Perry. Towards Optimal Distributed Consensus. In *FOCS*, 1989.
- [Bir85] Kenneth P. Birman. Replication and Fault-tolerance in the ISIS System. In *SOSP*, 1985.
- [Bir93] Kenneth P. Birman. The process group approach to reliable distributed computing. *Communications of the ACM*, 36, 1993.

- [BMC⁺15] Joseph Bonneau, Andrew Miller, Jeremy Clark, Arvind Narayanan, Joshua A. Kroll, and Edward W. Felten. SoK: Research Perspectives and Challenges for Bitcoin and Cryptocurrencies. In *IEEE S&P*, 2015.
- [BPR14] Carlos Eduardo Bezerra, Fernando Pedone, and Robbert Van Renesse. Scalable state-machine replication. In *DSN*, 2014.
- [Bra87] Gabriel Bracha. Asynchronous byzantine agreement protocols. *Inf. Comput.*, 75(2), November 1987.
- [BSA14] A. Bessani, J. Sousa, and E.E.P. Alchieri. State Machine Replication for the Masses with BFT-SMART. In *DSN*, 2014.
- [BT85a] Gabriel Bracha and Sam Toueg. Asynchronous consensus and broadcast protocols. *Journal of the ACM*, 32(4), 1985.
- [BT85b] Gabriel Bracha and Sam Toueg. Asynchronous Consensus and Broadcast Protocols. *Journal of the ACM*, 32(4), 1985.
- [BYFK08] Ziv Bar-Yossef, Roy Friedman, and Gabriel Kliot. RaWMS - Random Walk Based Lightweight Membership Service for Wireless Ad Hoc Networks. *ACM Trans. Comput. Syst.*, 26(2), 2008.
- [CCC⁺05] Manuel Costa, Jon Crowcroft, Miguel Castro, Antony Rowstron, Lidong Zhou, Lintao Zhang, and Paul Barham. Vigilante: End-to-end Containment of Internet Worms. In *SOSP*, 2005.
- [CDG⁺02] Miguel Castro, Peter Druschel, Ayalvadi Ganesh, Antony Rowstron, and Dan S Wallach. Secure routing for structured peer-to-peer overlay networks. *ACM SIGOPS Operating Systems Review*, 36(SI), 2002.
- [CDG⁺08] Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C. Hsieh, Deborah A. Wallach, Mike Burrows, Tushar Chandra, Andrew Fikes, and Robert E. Gruber. Bigtable: A Distributed Storage System for Structured Data. *ACM TOCS*, 26(2), 2008.
- [CDK⁺03] Miguel Castro, Peter Druschel, Anne-Marie Kermarrec, Animesh Nandi, Antony Rowstron, and Atul Singh. Splitstream: high-bandwidth multicast in cooperative environments. In *ACM SIGOPS Operating Systems Review*, volume 37, 2003.
- [CDKR02] Miguel Castro, Peter Druschel, A-M Kermarrec, and Antony IT Rowstron. Scribe: A large-scale and decentralized application-level multicast infrastructure. *Selected Areas in Communications, IEEE Journal on*, 20(8), 2002.
- [CGL⁺09] Yanpei Chen, Rean Griffith, Junda Liu, Randy H. Katz, and Anthony D. Joseph. Understanding TCP Incast Throughput Collapse in Datacenter Networks. In *WREN*, 2009.

Bibliography

- [CGR11] Christian Cachin, Rachid Guerraoui, and Lus Rodrigues. *Introduction to Reliable and Secure Distributed Programming*. Springer Publishing Company, Incorporated, 2nd edition, 2011.
- [CKV01] Gregory V. Chockler, Idit Keidar, and Roman Vitenberg. Group communication specifications: A comprehensive study. *ACM Comput. Surv.*, 33(4), 2001.
- [CL99] Miguel Castro and Barbara Liskov. Practical byzantine fault tolerance. In *Proceedings of the Third Symposium on Operating Systems Design and Implementation*, OSDI '99, 1999.
- [CL02] Miguel Castro and Barbara Liskov. Practical byzantine fault tolerance and proactive recovery. *ACM Transactions on Computer Systems*, 20(4), 2002.
- [CLM⁺08] Brendan Cully, Geoffrey Lefebvre, Dutch Meyer, Mike Feeley, Norm Hutchinson, and Andrew Warfield. Remus: High availability via asynchronous virtual machine replication. In *NSDI*, 2008.
- [CML⁺06] James Cowling, Daniel Myers, Barbara Liskov, Rodrigo Rodrigues, and Liuba Shrira. HQ replication: A hybrid quorum protocol for Byzantine fault tolerance. In *OSDI*, 2006.
- [CP02] Christian Cachin and Jonathan A. Poritz. Secure intrusion-tolerant replication on the internet. In *DSN*, 2002.
- [CSV17] Christian Cachin, Simon Schubert, and Marko Vukolic. Non-Determinism in Byzantine Fault-Tolerant Replication. In Panagiota Fatourou, Ernesto Jiménez, and Fernando Pedone, editors, *20th International Conference on Principles of Distributed Systems (OPODIS 2016)*, volume 70 of *Leibniz International Proceedings in Informatics (LIPIcs)*, Dagstuhl, Germany, 2017. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik.
- [CV17] Christian Cachin and Marko Vukolić. Blockchains consensus protocols in the wild. *arXiv preprint arXiv:1707.01873*, 2017.
- [CWA⁺09] Allen Clement, Edmund L Wong, Lorenzo Alvisi, Michael Dahlin, and Mirco Marchetti. Making Byzantine Fault Tolerant Systems Tolerate Byzantine Faults. In *NSDI*, 2009.
- [Dea09] Jeff Dean. Designs, lessons and advice from building large distributed systems. *Keynote from LADIS*, 2009.
- [DGH⁺88] Alan J. Demers, Daniel H. Greene, Carl Hauser, Wes Irish, John Larson, Scott Shenker, Howard E. Sturgis, Daniel C. Swinehart, and Douglas B. Terry. Epidemic algorithms for replicated database maintenance. *ACM SIGOPS Operating Systems Review*, 22(1), 1988.

- [DHR07] Danny Dolev, Ezra N. Hoch, and Robbert Renesse. Self-stabilizing and byzantine-tolerant overlay network. In *OPODIS*, 2007.
- [DLKA05] George Danezis, Chris Lesniewski-laas, M. Frans Kaashoek, and Ross Anderson. Sybil-resistant dht routing. In *In ESORICS*. Springer, 2005.
- [DM96] Danny Dolev and Dalia Malki. The Transis approach to high availability cluster communication. *Communications of the ACM*, 39(4), 1996.
- [DMS04] Roger Dingledine, Nick Mathewson, and Paul Syverson. Tor: The second-generation onion router. In *Proceedings of the 13th Conference on USENIX Security Symposium - Volume 13, SSYM'04*, Berkeley, CA, USA, 2004. USENIX Association.
- [Dou02] John R Douceur. The sybil attack. In *IPTPS*. Springer, 2002.
- [DRZ18] Sisi Duan, Michael K. Reiter, and Haibin Zhang. BEAT: Asynchronous BFT Made Practical. In *CCS*, 2018.
- [DS83] Danny Dolev and H. Raymond Strong. Authenticated algorithms for byzantine agreement. *SIAM J. Comput.*, 12(4), 1983.
- [DSW16] Christian Decker, Jochen Seidel, and Roger Wattenhofer. Bitcoin meets strong consistency. In *Proceedings of the 17th International Conference on Distributed Computing and Networking*, page 13, 2016.
- [EFGK03] Patrick Th Eugster, Pascal A Felber, Rachid Guerraoui, and Anne-Marie Kermarrec. The many faces of publish/subscribe. *ACM Computing Surveys (CSUR)*, 35(2), 2003.
- [EGSVR16] Ittay Eyal, Adem Efe Gencer, Emin Gün Sirer, and Robbert Van Renesse. Bitcoin-NG: A Scalable Blockchain Protocol. In *NSDI*, 2016.
- [FC97] Christof Fetzer and Flaviu Cristian. A fail-aware membership service. In *Reliable Distributed Systems*, 1997.
- [FLP85] Michael J. Fischer, Nancy A. Lynch, and Michael S. Paterson. Impossibility of distributed consensus with one faulty process. *Journal of the ACM*, 32(2), April 1985.
- [FSY05] Amos Fiat, Jared Saia, and Maxwell Young. Making Chord Robust to Byzantine Attacks. *Algorithms-ESA*, 2005.
- [GAG⁺18] Guy Golan-Gueta, Ittai Abraham, Shelly Grossman, Dahlia Malkhi, Benny Pinkas, Michael K. Reiter, Dragos-Adrian Seredinschi, Orr Tamir, and Alin Tomescu. SBFT: a scalable decentralized trust infrastructure for blockchains. *CoRR*, abs/1804.01626, 2018.

Bibliography

- [GBKA11] Lisa Glendenning, Ivan Beschastnikh, Arvind Krishnamurthy, and Thomas Anderson. Scalable consistency in Scatter. In *SOSP*, 2011.
- [GFG⁺09] Roxana Geambasu, Jarret Falkner, Paul Gardner, Tadayoshi Kohno, Arvind Krishnamurthy, and Henry M Levy. Experiences building security applications on DHTs. Technical report, Technical report, UW-CSE-09-09-01, 2009.
- [GGL03] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. The Google File System. In *SOSP*, 2003.
- [GHK13] Rachid Guerraoui, Florian Huc, and Anne-Marie Kermarrec. Highly dynamic distributed computing with byzantine failures. In *PODC*, 2013.
- [GHM⁺17] Yossi Gilad, Rotem Hemo, Silvio Micali, Georgios Vlachos, and Nickolai Zeldovich. Algorand: Scaling byzantine agreements for cryptocurrencies. In *SOSP*, 2017.
- [GKKZ11] Juan A Garay, Jonathan Katz, Ranjit Kumaresan, and Hong-Sheng Zhou. Adaptively Secure Broadcast, Revisited. In *PODC*. Citeseer, 2011.
- [GKL15] Juan Garay, Aggelos Kiayias, and Nikos Leonardos. The bitcoin backbone protocol: Analysis and applications. In *Advances in Cryptology - EUROCRYPT 2015*, 2015.
- [GKM⁺18] Rachid Guerraoui, Petr Kuznetsov, Matteo Monti, Matej Pavlovic, and Dragos-Adria Seredinschi. Asynchronous trustworthy transfers. *arXiv preprint arXiv:1812.10844*, 2018.
- [GLV06] Rachid Guerraoui, Ron R. Levy, and Marko Vukolic. Lucky read/write access to robust atomic storage. In *2006 International Conference on Dependable Systems and Networks (DSN 2006), 25-28 June 2006, Philadelphia, Pennsylvania, USA, Proceedings*, pages 125–136, 2006.
- [GPS18] Rachid Guerraoui, Matej Pavlovic, and Dragos-Adrian Seredinschi. Blockchain protocols: The adversary is in the details. In *Symposium on Foundations and Applications of Blockchain*, 2018.
- [Gup16] Saurabh Gupta. A Non-Consensus Based Decentralized Financial Transaction Processing Model with Support for Efficient Auditing. Master’s thesis, Arizona State University, USA, 2016.
- [GV06] Rachid Guerraoui and Marko Vukolic. How fast can a very robust read be? In *Proceedings of the Twenty-Fifth Annual ACM Symposium on Principles of Distributed Computing, PODC 2006, Denver, CO, USA, July 23-26, 2006*, pages 248–257, 2006.
- [GV07] Rachid Guerraoui and Marko Vukolic. Refined quorum systems. In *Proceedings of the Twenty-Sixth Annual ACM Symposium on Principles of Distributed Computing, PODC 2007, Portland, Oregon, USA, August 12-15, 2007*, pages 119–128, 2007.

- [GvR13] Haoyan Geng and Robbert van Renesse. Sprinkler - Reliable Broadcast for Geographically Dispersed Datacenters. In *Middleware 2013*. 2013.
- [Hea16] Mike Hearn. Corda: A distributed ledger. *Corda Technical White Paper*, 2016.
- [Her91] Maurice Herlihy. Wait-free synchronization. *ACM Transactions on Programming Languages and Systems*, 13(1), 1991.
- [HS00] Matti A Hiltunen and Richard D Schlichting. The cactus approach to building configurable middleware services. In *Proceedings of the Workshop on Dependable System Middleware and Group Communication (DSMGC 2000)*, 2000.
- [HT93] Vassos Hadzilacos and Sam Toueg. Fault-tolerant broadcasts and related problems. In Sape J. Mullender, editor, *Distributed Systems*, chapter 5. Addison-Wesley, 1993.
- [HW90] Maurice P Herlihy and Jeannette M Wing. Linearizability: A correctness condition for concurrent objects. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 12(3), 1990.
- [IRRS16] Damien Imbs, Sergio Rajsbaum, Michel Raynal, and Julien Stainer. Read/write shared memory abstraction on top of asynchronous byzantine message-passing systems. *J. Parallel Distrib. Comput.*, 93-94, 2016.
- [JAVR06] Håvard Johansen, André Allavena, and Robbert Van Renesse. Fireflies: scalable support for intrusion-tolerant network overlays. *ACM SIGOPS Operating Systems Review*, 40(4), 2006.
- [JF88] Colin J. Fidge. Timestamps in message-passing systems that preserve partial ordering. In *Proceedings of the 11th Australian Computer Science Conference*, 02 1988.
- [JT92] Prasad Jayanti and Sam Toueg. Some results on the impossibility, universality and decidability of consensus. In *International Workshop on Distributed Algorithms*, volume 647 of *LNCS*. Springer Verlag, 1992.
- [KAD⁺07] R. Kotla, L. Alvisi, M. Dahlin, A. Clement, and E. Wong. Zyzzyva: speculative byzantine fault tolerance. In *SOSP*, October 2007.
- [KBC⁺00] John Kubiatowicz, David Bindel, Yan Chen, Steven Czerwinski, Patrick Eaton, Dennis Geels, Ramakrishan Gummadi, Sean Rhea, Hakim Weatherspoon, Westley Weimer, Chris Wells, and Ben Zhao. Oceanstore: An architecture for global-scale persistent storage. *SIGPLAN Not.*, 2000.
- [KHG12] Aniket Kate, Yizhou Huang, and Ian Goldberg. Distributed key generation in the wild. *IACR Cryptology ePrint Archive*, 2012.

Bibliography

- [KJG⁺16] Eleftherios Kokoris Kogias, Philipp Jovanovic, Nicolas Gailly, Ismail Khoffi, Linus Gasser, and Bryan Ford. Enhancing bitcoin security and performance with strong consistency via collective signing. In *USENIX Security*, 2016.
- [KJW⁺18] Kolbeinn Karlsson, Weitao Jiang, Stephen Wicker, Danny Adams, Edwin Ma, Robbert van Renesse, and Hakim Weatherspoon. Vegvisir: A Partition-Tolerant Blockchain for the Internet-of-Things. In *ICDCS*, 2018.
- [KKJG⁺18] Eleftherios Kokoris-Kogias, Philipp Jovanovic, Linus Gasser, Nicolas Gailly, Ewa Syta, and Bryan Ford. Omniledger: A secure, scale-out, decentralized ledger via sharding. In *IEEE S&P*, 2018.
- [KLST11] Valerie King, Steven Lonargan, Jared Saia, and Amitabh Trehan. Load Balanced Scalable Byzantine Agreement through Quorum Building, with Full Information. In *International Conference on Distributed Computing and Networking*. Springer, 2011.
- [KRAV03] Dejan Kostić, Adolfo Rodriguez, Jeannie Albrecht, and Amin Vahdat. Bullet: High bandwidth data dissemination using an overlay mesh. In *SOSP*, 2003.
- [KT91] M Frans Kaashoek and Andrew S Tanenbaum. Group communication in the Amoeba distributed operating system. In *ICDCS*, 1991.
- [LAB⁺06] Jacob R. Lorch, Atul Adya, William J. Bolosky, Ronnie Chaiken, John R. Douceur, and Jon Howell. The smart way to migrate replicated stateful services. In *EuroSys*, 2006.
- [Lam78] Leslie Lamport. The implementation of reliable distributed multiprocess systems. *Computer Networks*, 2, 1978.
- [LCM⁺08] Harry C Li, Allen Clement, Mirco Marchetti, Manos Kapritsos, Luke Robison, Lorenzo Alvisi, and Mike Dahlin. FlightPath: Obedience vs. Choice in Cooperative Services. In *OSDI*, 2008.
- [LeM18] Colin LeMahieu. Nano: A feeless distributed cryptocurrency network. *Nano [Online resource]*. URL: <https://nano.org/en/whitepaper> (date of access: 18.01.2019), 2018.
- [LLK10] Chris Lesniewski-Lass and M Frans Kaashoek. Whanau: A sybil-proof distributed hash table. In *NSDI*, 2010.
- [LS03] Ching Law and Kai-Yeung Siu. Distributed construction of random expander networks. In *INFOCOM*, 2003.
- [LSP82] Leslie Lamport, Robert Shostak, and Marshall Pease. The byzantine generals problem. *ACM Trans. Program. Lang. Syst.*, 4(3), July 1982.

- [LYL⁺06] Zhiyu Liu, Ruifeng Yuan, Zhenhua Li, Hongxing Li, and Guihai Chen. Survive under high churn in structured P2P systems: evaluation and strategy. In *ICCS*, 2006.
- [Maz15] David Mazieres. The stellar consensus protocol: A federated model for internet-level consensus. *Stellar Development Foundation*, 2015.
- [MMR97] Dahlia Malkhi, Michael Merritt, and Ohad Rodeh. Secure reliable multicast protocols in a wan. In *ICDCS*, 1997.
- [MMR99] Dahlia Malkhi, Yishay Mansour, and Michael K. Reiter. On diffusing updates in a byzantine environment. In *SRDS*, 1999.
- [MMR00] Dahlia Malkhi, Michael Merritt, and Ohad Rodeh. Secure reliable multicast protocols in a wan. *Distributed Computing*, 13(1), Jan 2000.
- [MR97a] Dahlia Malkhi and Michael Reiter. Byzantine quorum systems. In *Proceedings of the twenty-ninth annual ACM symposium on Theory of computing*. ACM, 1997.
- [MR97b] Dahlia Malkhi and Michael K. Reiter. A high-throughput secure reliable multicast protocol. *Journal of Computer Security*, 5(2), 1997.
- [MRWW01] Dahlia Malkhi, Michael K Reiter, Avishai Wool, and Rebecca N Wright. Probabilistic quorum systems. *Inf. Comput.*, 170(2), November 2001.
- [MS03] Yaron M Minsky and Fred B Schneider. Tolerating malicious gossip. *Distributed Computing*, 16(1), 2003.
- [MXC⁺16] Andrew Miller, Yu Xia, Kyle Croman, Elaine Shi, and Dawn Song. The honey badger of bft protocols. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*. ACM, 2016.
- [Nak08] Satoshi Nakamoto. Bitcoin: A peer-to-peer electronic cash system, 2008.
- [NDO11] Edmund B. Nightingale, John R. Douceur, and Vince Orgovan. Cycles, Cells and Platters: An Empirical Analysis of Hardware Failures on a Million Consumer PCs. In *EuroSys*, 2011.
- [OGP03] David Oppenheimer, Archana Ganapathi, and David A. Patterson. Why do Internet services fail, and what can be done about it? In *USITS*, 2003.
- [OO14] Diego Ongaro and John Ousterhout. In search of an understandable consensus algorithm. *USENIX ATC*, 2014.
- [PS02] Fernando Pedone and André Schiper. Handling message semantics with generic broadcast protocols. *Distributed Computing*, 15(2), 2002.

Bibliography

- [RDJ13] Roberto Roverso, Jonathon Dowling, and Mark Jelasity. Through the wormhole: Low cost, fresh peer sampling for the internet. In *Peer-to-Peer Computing (P2P)*, 2013.
- [REG⁺03] Sean C Rhea, Patrick R Eaton, Dennis Geels, Hakim Weatherspoon, Ben Y Zhao, and John Kubiawicz. Pond: The OceanStore prototype. In *FAST*, volume 3, 2003.
- [RGRK04] Sean Rhea, Dennis Geels, Timothy Roscoe, and John Kubiawicz. Handling churn in a DHT. In *USENIX*, 2004.
- [RL03] Rodrigo Rodrigues and Barbara Liskov. Rosebud: A scalable byzantine-fault-tolerant storage architecture. Technical Report MIT-LCS-TR-932 and MIT-CSAIL-TR-2003-035, 2003.
- [RLGS14] Philip Rapoport, Roman Leal, Patrick Griffin, and Wellingtona Sculley. The Ripple Protocol, 2014.
- [RLS02] Rodrigo Rodrigues, Barbara Liskov, and Liuba Shrira. The design of a robust peer-to-peer system. In *ACM SIGOPS European workshop: beyond the PC - EW10*, 2002.
- [SBV18] Joao Sousa, Alysson Bessani, and Marko Vukolic. A byzantine fault-tolerant ordering service for the hyperledger fabric blockchain platform. In *DSN*, 2018.
- [Sch90] Fred B. Schneider. Implementing fault-tolerant services using the state machine approach: A tutorial. *ACM Comput. Surv.*, 22(4), 1990.
- [Sch05] Christian Scheideler. How to Spread Adversarial Nodes? Rotate! In *STOC*. ACM, 2005.
- [SDG10] Bianca Schroeder, Sotirios Damouras, and Phillipa Gill. Understanding latent sector errors and how to protect against them. In *FAST*, 2010.
- [SHPG12] Ankit Singla, Chi-Yao Hong, Lucian Popa, and Philip Brighten Godfrey. Jellyfish: Networking data centers randomly. In *NSDI*, 2012.
- [SMK⁺01] Ion Stoica, Robert Morris, David Karger, M Frans Kaashoek, and Hari Balakrishnan. Chord: A scalable peer-to-peer lookup service for internet applications. In *SIGCOMM*, 2001.
- [SPBZ11] Marc Shapiro, Nuno Preguiça, Carlos Baquero, and Marek Zawirski. Conflict-free replicated data types. In *Stabilization, Safety, and Security of Distributed Systems*. Springer, 2011.
- [sql] <https://www.sqlite.org/>.

-
- [Sza97] Nick Szabo. Formalizing and securing relationships on public networks. *First Monday*, 2(9), 1997.
- [Tou84] Sam Toueg. Randomized byzantine agreements. In *Proceedings of the Third Annual ACM Symposium on Principles of Distributed Computing*, PODC '84, New York, NY, USA, 1984. ACM.
- [TR18] Team-Rocket. Snowflake to Avalanche: A Novel Metastable Consensus Protocol Family for Cryptocurrencies. *White Paper*, 2018. Revision: 05/16/2018 21:51:26 UTC.
- [vdHLZZ15] Jelle van den Hooff, David Lazar, Matei Zaharia, and Nikolai Zeldovich. Vuvuzela: Scalable private messaging resistant to traffic analysis. In *Proceedings of the 25th Symposium on Operating Systems Principles*, SOSP '15, New York, NY, USA, 2015. ACM.
- [VGLN07] Long Vu, Indranil Gupta, Jin Liang, and Klara Nahrstedt. Measurement and modeling of a large-scale overlay for multimedia streaming. In *QSHINE*, 2007.
- [VRBM96] Robbert Van Renesse, Kenneth P Birman, and Silvano Maffeis. Horus: A Flexible Group Communication System. *Communications of the ACM*, 39(4), 1996.
- [Vuk10] Marko Vukolic. The origin of quorum systems. *Bulletin of the EATCS*, 101, 2010.
- [Vuk15] Marko Vukolić. The Quest for Scalable Blockchain Fabric: Proof-of-work vs. BFT Replication. In *International Workshop on Open Problems in Network Security*. Springer, 2015.
- [VvS13] Spyros Voulgaris and Maarten van Steen. Vicinity: A pinch of randomness brings out the structure. In David Ebers and Karsten Schwan, editors, *Middleware*, Berlin, Heidelberg, 2013. Springer Berlin Heidelberg.
- [WECK07] Hakim Weatherspoon, Patrick Eaton, Byung-Gon Chun, and John Kubiatowicz. Antiquity: exploiting a secure log for wide-area distributed storage. *ACM SIGOPS Operating Systems Review*, 41(3), 2007.
- [Woo15] Gavin Wood. Ethereum: A secure decentralized generalized transaction ledger. White paper, 2015.
- [YKGK10] Maxwell Young, Aniket Kate, Ian Goldberg, and Martin Karsten. Practical Robust Communication in DHTs Tolerating a Byzantine Adversary. In *ICDCS*, 2010.
- [ZHS⁺04] Ben Y Zhao, Ling Huang, Jeremy Stribling, Sean C Rhea, Anthony D Joseph, and John D Kubiatowicz. Tapestry: A resilient global-scale overlay for service deployment. *IEEE JSAC*, 22(1), 2004.

Matej Pavlovič

EPFL IC IINFCOM DCL, INR 314 (Bâtiment INR), Station 14, CH-1015 Lausanne, Switzerland
matej.pavlovic@epfl.ch • +41 76 650 7980 • <https://people.epfl.ch/matej.pavlovic>

RESEARCH INTERESTS

Large-scale dynamic distributed systems, State machine replication, Byzantine fault tolerance, Blockchain, Distributed storage, Consistency

EDUCATION

École Polytechnique Fédérale de Lausanne (EPFL), Lausanne, Switzerland

- PhD candidate, Computer Science Sep 2013 – Sep 2019
 - Advisor: Prof. Rachid Guerraoui
 - Thesis title: “Scaling Byzantine Fault Tolerance”
- Internship on large-scale dynamic distributed BFT systems Oct 2012 – Jul 2013
- ERASMUS Exchange Sep 2011 – Jun 2012
 - Master’s thesis: “Implementation of a Distributed Computation Framework”
 - Thesis supervisors: Prof. Rachid Guerraoui (EPFL), Prof. Ulrich Schmid (TU Wien)

Vienna University of Technology (TU Wien), Vienna, Austria

Oct 2007 – Jun 2011

- Master of Science (MSc) in Computer Engineering
 - Graduated with distinction, GPA: 1.176 (scale from 1 to 5)
- Bachelor of Science (BSc) in Computer Engineering
 - Graduated with distinction, GPA: 1.383 (scale from 1 to 5)

INTERNSHIP

Oracle Labs East, Boston (Burlington), Massachusetts, USA

Jun 2017 – Sep 2017

- Penumbra and Scalable Synchronization Research Group (now Distributed Systems Group)
- Persistent memory key-value store, persistent multi-word compare-and-swap

CONFERENCE AND JOURNAL PUBLICATIONS

- Rachid Guerraoui, Petr Kuznetsov, Matteo Monti, Matej Pavlovic, Dragos-Adrian Seredinschi
“The Consensus Number of a Cryptocurrency”
PODC 2019
- Oana Balmau, Rachid Guerraoui, Anne-Marie Kermarrec, Alexandre Maurer, Matej Pavlovič, Willy Zwaenepoel
“The Fake News Vaccine”
NETYS 2019
- Matej Pavlovič, Alex Kogan, Virendra J. Marathe, Tim Harris
“Brief Announcement: Persistent Multi-Word Compare-and-Swap”
PODC 2018.
- Yihe Huang, Matej Pavlovič, Virendra Marathe, Margo Seltzer, Tim Harris, Steve Blyan
“Closing the Performance Gap Between Volatile and Persistent Key-Value Stores Using Cross-Referencing Logs”
USENIX ATC 2018.
- Rachid Guerraoui, Matej Pavlovič, Dragos-Adrian Seredinschi
“Blockchain Protocols: The Adversary is in the Details”
Symposium on Foundations and Applications of Blockchain 2018
- Rachid Guerraoui, Matej Pavlovič, Dragos-Adrian Seredinschi
“Incremental Consistency Guarantees for Replicated Objects”
OSDI 2016
- Rachid Guerraoui, Anne-Marie Kermarrec, Matej Pavlovič, Dragos-Adrian Seredinschi
“Atum: Scalable Group Communication Using Volatile Groups”
Middleware 2016
- Rachid Guerraoui, Matej Pavlovič, Dragos-Adrian Seredinschi
“Trade-offs in Replicated Systems”
IEEE Data Engineering Bulletin 39(1) 2016

- OTHER PUBLICATIONS**
- Rachid Guerraoui, Petr Kuznetsov, Matteo Monti, Matej Pavlovič, Dragos-Adrian Seredinschi
“AT2: Asynchronous Trustworthy Transfers”
<https://arxiv.org/abs/1812.10844>, 2018
- SKILLS**
- Programming
 - Python
 - Java
 - JavaScript
 - C
 - Circuit design in VHDL
 - Microcontroller Programming
- AWARDS AND SCHOLARSHIPS**
- EPFL IC Faculty Award for Teaching Excellence Dec 2017
 - Mondri Austria Student Scholarship Oct 2007 – Aug 2013
 - For exceptional achievements in the education so far and extra-curricular activities
 - Full coverage of studying and living cost during undergraduate and master’s studies
- TEACHING & MENTORING**
- TEACHING**
- Distributed Algorithms (CS-451) - master’s class, EPFL Fall 2018
 - Industrial Automation (CS-487) - master’s class, EPFL Spring 2018
 - Information, Computation, Communication (CS-110(b)) - undergrad class, EPFL Fall 2017
 - Industrial automation (CS-487) - master’s class, EPFL Spring 2017
 - Information, Computation, Communication (CS-110(b)) - undergrad class, EPFL Fall 2016
 - Mathématiques générales II (UNIL-108) - undergrad class, UNIL Spring 2016
 - Information, Computation, Communication (CS-110(b)) - undergrad class, EPFL Fall 2015
 - Mathematics II (MATH-186) - undergrad class, UNIL Spring 2015
 - Distributed algorithms (CS-451) - master’s class, EPFL Fall 2014
 - Probabilities and statistics (MATH-232) - undergrad class, EPFL Spring 2014
- MENTORING**
- *EDIC doctoral project*: Zeinab Shmeis - Evaluation of a Scalable Secure Broadcast Protocol (Fall 2018)
 - *Semester project*: Iva Najdenova - Improving the Efficiency of a Blockchain Protocol (Spring 2018)
 - *Semester project*: Florian Alonso, Dennis van der Bij - Large Scale Gossip Based State Machine Replication (Spring 2018)
 - *Semester project*: André Baptista Águas, Artem Shevchenko - Collaborative Blockchain Reinforcement in Blockchain Systems. (Spring 2017)
 - *Semester project*: Radmila Popovic - Application of incremental consistency guarantees in a social network application. (Summer 2016)
 - *Master’s thesis*: Kenji Relut - Implementation of Incremental Consistency in global-scale storage systems with Zookeeper’s queue (Spring 2016)
 - *Summer internship*: Mahammad Valiyev - Implementation of a Byzantine fault tolerant computation framework (Summer 2015)
 - *Semester project*: Jean Ashton - Design and implementation of a real-time multiplayer browser game (Spring 2015)
- LANGUAGES**
- Slovak: native language
 - English: fluent (speaking / writing)
 - German: fluent (speaking / writing)
 - French: fluent in speaking, intermediate in writing
- EXTRA-CURRICULAR ACTIVITIES**
- Judo - 3x national champion, trainer and referee in Slovakia
 - Swing dancing
 - Outdoor activities - ski touring, snowboarding, mountaineering, rock climbing, mountain biking, cycling
 - Piano

